Sistemas Informáticos

Curso 2008 - 2009

Máquina Virtual sobre procesador ARM

Alumnos:

Roberto Cano Fernández

Luis Óscar Madrid Rico

Sergio del Valle Salvador

Directora de proyecto:

Katzalin Olcoz Herrero

Facultad de Informática

Universidad Complutense de Madrid

Autorización

Autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Roberto Cano Fernández

Luis Oscar Madrid Rico

Sergio del Valle Salvador

Agradecimientos

Queremos agradecer de forma especial a nuestra profesora Katzalin Olcoz Herrero por su atención, apoyo y paciencia. A Luis Piñuel y Manuel Prieto por su ayuda desinteresada en los momentos clave. Por último queremos agradecer a nuestros familiares y amigos su comprensión y apoyo en todo momento, sin el cual no hubiésemos podido llevar a buen puerto este proyecto.

Palabra clave

Virtualización, paravirtualización, virtualización a nivel del sistema operativo, contenedor, jaula, emulación, sandbox.

Resumen

La idea de virtualización en el mundo de la informática está ampliamente extendida pero en el campo de los sistemas empotrados y más concretamente para arquitecturas ARM está dando sus primeros pasos. Por ello, se decidió desarrollar una máquina virtual sobre ARM de manera que podamos ser pioneros en este sector. Esta máquina virtual está implementada a nivel de sistema operativo y en ella se trata de crear una serie de jaulas o contenedores que tendrán su propio sistema de ficheros, gracias a las características que proporciona la función chroot, y que también tendrán aislados los procesos del interior con respecto de los procesos del exterior del contenedor.

A lo largo de este documento encontrará las investigaciones realizadas previas al desarrollo y un resumen del concepto de virtualización donde también se explican las razones por las cuales se tomaron las decisiones que llevaron al modelo de "jaula". Posteriormente está detallado todo el desarrollo dividido en los apartados de soluciones descartadas, implementación y líneas de trabajo futuras. Por último, se encuentran una serie de análisis de la implementación y de las pruebas realizadas y una conclusión de todo el trabajo realizado a lo largo del curso.

The idea of virtualization in the computer's science world is widely extended although in the field of the embedded systems, in fact ARM architectures, it is in its first steps. Therefore, we decided to develop a virtual machine for ARM so that we could be pioneers in this sector. This virtual machine monitor is implemented at an operative system level in which we've tried to create a number of jails or containers that would have their own file system, due to the functionality of chroot, and also the inside processes would be isolated from the outside processes.

Throughout this document you will find the predevelopment investigations done and a summary of the virtualization concept where the reasons which led to the decision of the "jail" model are also explained. Afterwards, the entire development is detailed divided into three parts called discarded solutions, implementation and future work lines. By the end, there is a number of analysis about the implementation and the benchmarks we have done and the conclusions of the work we have done through the year.

Indice

Introducción

Hoy en día el concepto de virtualización está muy extendido por su demostrada utilidad en todos los ámbitos de la informática, desde un usuario que virtualiza un sistema operativo para utilizar cierta aplicación que no tiene disponible en su entorno habitual, pasando por sistemas operativos con propósito educativo (por ejemplo Minix), hasta empresas que trabajan con máquinas virtuales ante la imposibilidad de modificar el servidor remoto donde finalmente se van a ejecutar sus aplicaciones.

El uso masivo de este conjunto de tecnologías favorece la competitividad en el mercado. Efectivamente existen multitud de empresas que se dedican a desarrollar entornos para la virtualización de recursos. Más tarde hablaremos de algunas de ellas y de sus productos.

Nuestra elección como proyecto de Sistemas Informáticos de investigar éste conjunto de tecnologías, responde a la evolución que presenta el mercado. Conforme se vaya consolidando la virtualización como una solución a cierto tipo de problemas el número de proyectos de investigación y la cantidad de dinero dedicado en el sector crecerá.

Motivaciones

En la actualidad, la virtualización sobre cualquier plataforma se encuentra en plena expansión. En un futuro será una de las tecnologías predominantes y estará integrada en multitud de sistemas. Bajo esta perspectiva nace nuestra curiosidad sobre este mundo.

Nos centramos en los procesadores ARM porque estos procesadores se encuentran entre los más punteros del mercado de los sistemas empotrados. El ARM 7 es el más usado por su bajo coste y altas prestaciones (es un procesador RISC con suficiente potencia para la mayoría de aplicaciones en ese tipo de entornos). Además, el modelo de licencias que la empresa desarrolladora de los procesadores emplea ha facilitado su extensión a una gran variedad de dispositivos (desde calculadoras hasta teléfonos móviles, pasando por lavadoras o televisiones).

Como resultado de ello, surgieron una serie de propuestas que terminó desembocando en el proyecto que actualmente presentamos.

Usos actuales

Los posibles usos de las máquinas virtuales son variados, algunos actuales y otros que se irán desarrollando en el futuro. A continuación realizamos una breve descripción:

- **Prueba de nuevos sistemas:** es habitual en aplicaciones de escritorio no desear que probarla signifique poner en peligro la estabilidad del sistema. Una máquina virtual donde se lleve a cabo esta prueba suele ser una solución a este problema.
- **Desarrollo de aplicaciones:** en ocasiones, el desarrollo de un nuevo producto debe realizarse de forma simultánea mientras el producto está en uso. Esta situación puede ser evitada encapsulando el entorno de desarrollo en una máquina virtual que evite modificar máquina donde se está ejecutando el producto. También puede usarse para desarrollar aplicaciones multiplataforma o para comprobar el correcto funcionamiento del programa.
- **Aplicaciones obsoletas y transiciones**: ciertas aplicaciones son exclusivas para algunos sistemas operativos por lo que en caso de querer utilizarlas tendremos dos opciones: instalar el sistema operativo requerido, o ejecutarlo sobre una máquina virtual. Últimamente, se está investigando la integración de máquinas virtuales directamente en el kernel para mantener la compatibilidad de programas. Además, permite la migración de aplicaciones a una nueva arquitectura de manera progresiva, realizándose primero sobre un entorno controlado como es una máquina virtual.
- **Servidores**: el uso de máquinas virtuales permite proporcionar multitud de servicios en un mismo servidor de manera segura. El aislamiento que proporciona la máquina virtual permite la ejecución simultánea de diversos entornos de manera transparente al usuario.

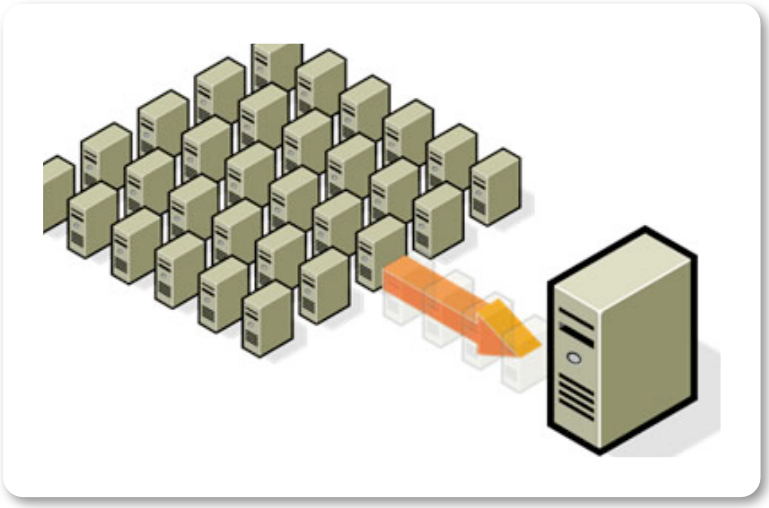


Ilustración 1: virtualización en servidores (idea)

- **Seguridad**: el uso de virtualización nos da otro nivel de seguridad más, ya que el sistema operativo virtualizado no tiene acceso directo al hardware sino que el sistema anfitrión se encarga de acceder a la E/S o a los servicios de red por ejemplo. Además, la encapsulación del sistema permite controlar sus acciones. También puede ser útil en entornos donde una

aplicación puede poner en riesgo la integridad del resto del sistema por diversos motivos (seguridad, estado de desarrollo poco avanzado, etc.).

Además, en los próximos años el papel que juega la virtualización en la integración de sistemas debería crecer, llegando a ser una pieza clave en multitud de arquitecturas como lo demuestra el hecho de que las principales arquitecturas ya incluyen soporte hardware para la virtualización.

Objetivos

El objetivo principal ha sido implementar una máquina virtual sobre el procesador ARM que permita la ejecución aislada de grupos de procesos de manera que no puedan interactuar con otros procesos ni con el sistema operativo nativo sin un cierto control.

Nuestros objetivos iniciales son los de cualquier máquina virtual:

- Seguridad
- Eficiencia
- Transparencia

Para cumplirlos hemos preparado un entorno controlado al que de ahora en adelante llamaremos jaula, donde hemos tratado de aislar los procesos y hemos ejecutado unos test de rendimiento para comprobar si se producía mucha degradación al ejecutarlos dentro de la jaula.

Recorrido a través del documento

En primer lugar, hemos investigado algunas de las tecnologías existentes actualmente en el mercado. Todas ellas serán explicadas con detalle en la sección “Investigaciones previas”.

En el apartado de “Virtualización” hablaremos sobre los modelos más usados de virtualización, algunas de las aplicaciones que lo usan, así como los productos software existentes en la plataforma ARM que permitan virtualizar diversos sistemas operativos.

También hemos reservado un espacio para comentar algunos descartes en nuestra solución que han ocurrido por diversos motivos. Son o pudieron ser parte de la implementación final pero en algún momento se decidió que no fuera así. Hablaremos de algunos de ellos en el apartado “Soluciones descartadas”, el resto se podrían llegar a desarrollar con más tiempo y los comentaremos en “Líneas de trabajo futuras”.

Nuestra solución, requiere de una preparación adecuada del entorno donde se ejecutará la jaula. El capítulo “Preparación del entorno” se ocupa de explicar los pasos necesarios para ello. Se

realizarán además, recomendaciones basadas en nuestra propia experiencia para un correcto funcionamiento de nuestra solución.

A continuación, en las secciones “Modelo de desarrollo” e “Implementación” explicaremos detalladamente nuestra solución. Enumeraremos, primero las fases de desarrollo por las que hemos pasado hasta llegar al diseño actual, además de hablar en profundidad de las modificaciones al kernel de Linux realizadas y qué esperamos obtener a través de esas modificaciones. Comentaremos brevemente algunas posibles modificaciones que no hemos llevado a cabo.

Realizaremos un análisis de nuestra solución en “Análisis y comentarios”. Expondremos algunos test que hemos llevado a cabo y propondremos una batería de pruebas para concluir que se han obtenido los resultados esperados (en algunos casos).

Terminaremos con la sección “Conclusiones” donde hablaremos de aspectos que nos han resultado gratos, así como de otros aspectos que no nos han satisfecho, nuestro problemas durante el desarrollo, consejos si se quiere realizar un trabajo parecido (de la experiencia se aprende) y finalizaremos con las posibles líneas de trabajo futuras.

Plataforma

En primer lugar hablaremos del hardware para el que se ha realizado la implementación, para más tarde comentar las opciones de sistemas operativos que nos planteamos. Haremos énfasis en algunos puntos que consideramos relevantes.

Hardware

Los procesadores ARM usan el modelo de arquitectura *RISC* (Reduced Instruction Set Computer) que reduce considerablemente la complejidad del repertorio de instrucciones y por tanto su descodificación, permitiendo, generalmente, un mayor rendimiento que los procesadores de arquitectura *CISC* (Complex Instruction Set Computer). La familia de procesadores ARM tienen como objetivo obtener un alto rendimiento para un bajo consumo energético y un tamaño mínimo siendo, por tanto, los procesadores más utilizados en el mercado de los sistemas empotrados.

Aunque en un principio la versión de procesadores ARM que íbamos a usar es ARM7 TDMI. Más tarde, una vez optamos por la solución de "virtualización a nivel de sistema operativo" y decidimos realizar la implementación sobre Linux, la importancia del hardware disminuyó, pues nos resultaba sencillo portar nuestros resultados a otra arquitectura. Finalmente, Luis Piñuel nos prestó una placa BeagleBoard sobre la que realizar las últimas pruebas.

Aún así, consideramos imprescindible explicar la arquitectura del ARM 7, pues todas las sucesivas versiones de procesadores ARM expanden el concepto implementado por este núcleo (mejoras en el repertorio de instrucciones, más etapas del Pipeline, diversos módulos adicionales, etc).

ARM 7

El procesador ARM 7 y más concretamente el ARM7TDMI pertenece a la familia de microprocesadores de 32 bits de propósito general de ARM.

El ARM7TDMI está segmentado en tres etapas de la siguiente manera:

- **Fetch**: búsqueda de la instrucción en memoria
- **Decode**: decodificación de los registros a usar en la instrucción.
- **Execute**: lectura del banco de registros, realización de la operación en la ALU y escritura en el banco de registros.

El ARM7TDMI tiene una arquitectura Von Neumann, con un único bus de datos de 32 bits de longitud que transporta tanto instrucciones como datos. Sólo las instrucciones load, store y swap pueden acceder a los datos desde memoria. Estos datos pueden tener una longitud de 8, 16 o 32

bits. La interfaz está diseñada para permitir mejoras potenciales mientras se minimiza el uso de la memoria. Además, permite la integración en dispositivos con coprocesadores.

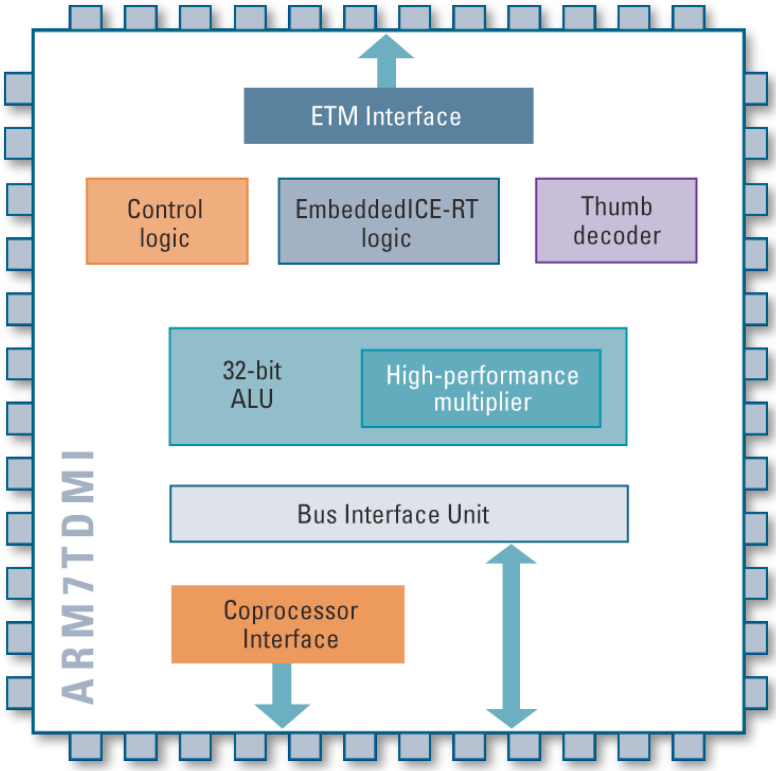


Ilustración 2: distribución del encapsulado del ARM 7 TDMI

Los procesadores ARM tienen un hardware adicional llamado EmbeddedICELogic que, haciendo las veces de *depurador* ofrece una serie de herramientas software para poner a punto código que se ejecute sobre el procesador.

El ARM7 TDMI tiene dos repertorios de instrucciones diferentes:

- Repertorio ARM de 32 bits.
- Repertorio *Thumb* de 16 bits.

Thumb implementa un repertorio de instrucciones de 16 bits en una arquitectura de 32 bits para mejorar el rendimiento de una arquitectura de 16 bits y obtener una mayor densidad de código que en una arquitectura de 32 bits. Las instrucciones Thumb son transformadas, de manera transparente para el programador, en instrucciones de 32 bits en tiempo de ejecución sin pérdida de rendimiento. Por tanto, Thumb tiene todas las ventajas de un núcleo de 32 bits, haciendo que el núcleo del ARM7TDMI sea ideal para aplicaciones de sistemas empotrados que tienen una gran restricción de ancho de banda en memoria.

La disponibilidad de un repertorio de 16 bits y otro de 32 permite a los diseñadores tener flexibilidad entre el tamaño del código y el rendimiento a nivel de subrutinas dependiendo de las necesidades de su aplicación.

ARM Cortex-A8

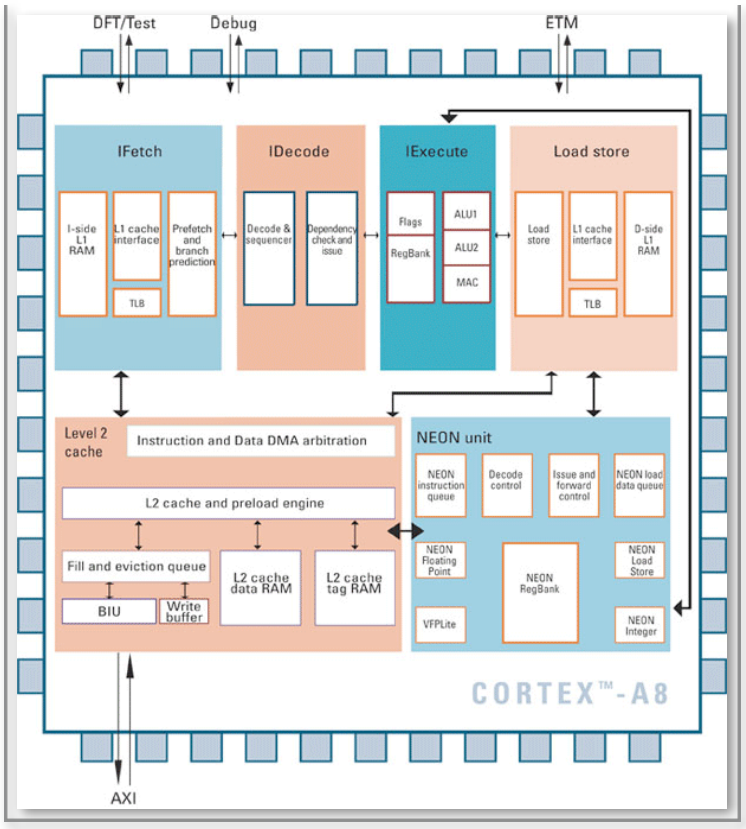


Ilustración 4: ARM Cortex A8 (core)

Basado en la arquitectura ARMv7, el procesador ARM Cortex-A8 presenta las siguientes características y mejoras sobre versiones anteriores:

- **NEON Media Processing Engine:** acelera la comunicación y las funciones de proceso de señales para incrementar el rendimiento específico de las aplicaciones. También proporciona una aceleración significativa en operaciones en Punto Flotante para ambas precisiones, simple y doble. Dobla el rendimiento de la anterior ARM FPU.
- **Mejora de la Cache de nivel 1:** la mejora en la cache de nivel 1 en rendimiento y consumo combina técnicas de mínima latencia de acceso para maximizar el rendimiento y minimizar el consumo de energía. También proporciona la opción para la coherencia de cache en la comunicación entre procesadores así como admite el uso de sistemas operativos multi-proceso corriendo sobre varios núcleos.
- **Tecnología Thumb-2.**

- **Tecnología TrustZone:** asegura una implementación fiable de aplicaciones de seguridad en una gama que comprende desde gestión de derechos digitales hasta pago electrónico.
- **Tecnología Jazelle:** proporciona una reducción de hasta un tercio del tamaño del código en tiempo de ejecución y en tiempo de compilación de lenguaje Java y otros lenguajes soportados.

Sólo podemos terminar haciendo hincapié en que el Cortex A-8(y en general la familia Cortex) es un de los procesadores de bajo consumo más avanzados que existen en el mercado.

BeagleBoard

Ilustración 3: Placa BeagleBoard

La placa BeagleBoard es una placa "low-cost" fabricada por la empresa de nombre homónimo que sin embargo ofrece una serie de funcionalidades muy interesantes y poco habituales en esta línea de productos.

En primer lugar, su chip principal es el OMAP 3530 cuyas características principales se enumeran a continuación:

- Usa un procesador ARM Cortex A-8 a 600 MHz con caché de nivel 2. Esto le dota de una potencia inusitada en este tipo de dispositivos.
- Dispone de un acelerador gráfico POWERVR SGX530, lo que le permite ser capaz de acelerar OpenGL© 2.0 tanto en 2D como en 3D.
- También incluye en el mismo encapsulado un DSP orientado a vídeo TMS320C64x+ a 430 MHZ.
- Consumo energético muy bajo, lo que hace innecesaria refrigeración adicional.

Además, la placa tiene integrada una serie de conexiones que permiten conectarla a periféricos externos como monitores digitales (mediante su conexión DVI-D), cualquier dispositivo que permita conectarse mediante USB(teclados, WiFi, Bluetooth…), audio, cargadores USB, así como también permite leer tarjetas SD o MMC+.

Software

Un sistema operativo de tiempo real (RTOS -*Real Time Operating System* en inglés), es un sistema operativo que ha sido desarrollado para aplicaciones de tiempo real. Como tal, se le exige corrección en sus respuestas bajo ciertas restricciones de tiempo. Si no las respeta, se dirá que el sistema ha fallado. Para garantizar el comportamiento correcto en el tiempo requerido se necesita que el sistema sea predecible (determinista).

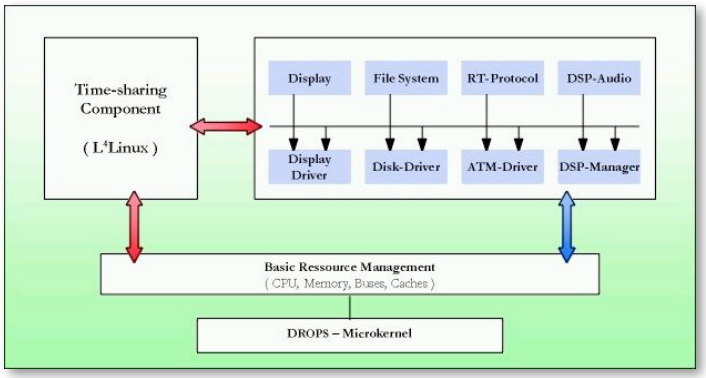


Ilustración 5: Diagrama de un SO de tiempo real

Diversos desarrollos de RTOS fueron analizados y estudiados en su momento. Mencionamos aquí algunos de ellos.

eCos

Diseñado para sistemas en tiempo real empotrados, eCos implementa una arquitectura clásica multi-threaded con un amplio conjunto de elementos de sincronización. Esto proporciona tiempos de respuesta predecibles, latencias mínimas de interrupciones y una gama baja de cambios de contexto.

Destacamos la capacidad de adaptación de eCos, demostrada por los ports de eCos a prácticamente todas las arquitecturas modernas de 32 bits entre las que se encuentran ARM, Coldfire/68K, Hitachi SH2/3/4, Intel x86, MIPS, PowerPC y SPARC. El cargador de arranque RedBoot y los complementos de depuración están también basados en el HAL de eCos proporcionando ambos una solución a la depuración vía software y una portabilidad del sistema en un sólo paso de desarrollo.

FreeRTOS

Es un desarrollo OpenSource conocido por su fiabilidad y simplicidad. Además, su código se encuentra únicamente en tres ficheros (el resto son ficheros de configuración) lo que facilita su portabilidad. Se publicita como verdaderamente gratis para aplicaciones comerciales y dispone de un programa de confidencialidad para este fin.

µCLinux

El sistema µCLinux original era un derivado del kernel 2.0 de Linux hecho para micro controladores sin MMU's (Unidad de Gestión de Memoria). Sin embargo, el proyecto Linux/ Microcontroller ha crecido pasando a ser un desarrollo prácticamente independiente y reconocido. µClinux como sistema operativo actual incluye las versiones del kernel de Linux 2.0, 2.4 y 2.6 así como una colección de aplicaciones de usuario, librerías(incluida su propia versión de *libc*) y listas de herramientas.

En un principio, cuando el desarrollo iba a ser sobre un procesador ARM7, µClinux se convirtió en nuestra primera opción, pero fue finalmente descartado por un núcleo Linux habitual por los motivos comentados al principio del capítulo.

Linux

Hacemos mención expresa de este sistema operativo porque fue finalmente el seleccionado para implementar nuestro proyecto, aunque no consideramos necesario hacer ningún comentario adicional debido a su fama y a su creciente uso tanto en escritorio como en dispositivos empotrados.

Virtualización

En este capítulo comentaremos los distintos modelos de virtualización y máquinas virtuales que existen. También hablaremos sobre las soluciones de virtualización existentes sobre plataforma ARM. En primer lugar comentaremos brevemente aquellos que no hemos usado.

Modelos de virtualización no usados

Tenemos distintos enfoques a la hora de afrontar el problema de la virtualización. A continuación expondremos los que consideramos menos útiles para nuestro proyecto: emuladores y virtualización completa.

Un **emulador** es un software que permite ejecutar programas en una arquitectura hardware diferente de aquella para la que fueron escritos originalmente. A diferencia de un simulador, que sólo trata de reproducir el comportamiento del programa, un emulador trata de modelar de forma precisa el dispositivo que se está emulando. Consiste en crear una capa de abstracción, pero ejecutando instrucciones en una máquina del mismo tipo, y da como resultados obtener una computadora dentro de otra.

Virtualización completa es en donde la máquina virtual simula un hardware suficiente para permitir un sistema operativo invitado sin modificar (uno diseñado para la misma CPU) para ejecutarse de forma aislada. Este enfoque fue el pionero en 1966 con CP-40 y CP[-67]/CMS, predecesores de la familia de máquinas virtuales de IBM.

Ahora recorreremos el resto de alternativas entre las cuales se encuentra la virtualización a nivel de sistema operativo que es la que hemos escogido nosotros. Comenzaremos con la definición de Popek y Goldberg, que es la primera formalización de lo que necesita ser una máquina virtual para ser considerada tal.

Virtualización según Popek & Goldberg

Popek y Goldberg (en adelante P&G) contribuyeron a la definición formal y seria siendo los primeros en plantear de forma estructurada los requisitos que debía tener una máquina virtual. Estos requisitos debían ser cumplidos no sólo para ser considerada en estos términos, sino para ser considerada una *buena* máquina virtual.

En primer lugar propusieron una separación entre el sistema “host”, que sería el sistema base sobre el que correrían aplicaciones y máquinas virtuales, el “hipervisor”, que se encargaría de gestionar las máquinas virtuales y los sistemas “guest”, que serían las máquinas virtuales propiamente dichas.

Sus relaciones se pueden expresar de la siguiente manera:

- **Equivalencia:** un programa o aplicación corriendo sobre la máquina virtual debería demostrar un comportamiento esencialmente idéntico que el demostrado en el equivalente sobre la máquina directamente.
- **Control de recursos**: el hipervisor debe estar en completo control de los recursos virtualizados.
- **Eficiencia:** una fracción estadísticamente dominante de instrucciones máquina deben ser ejecutadas sin la intervención del hipervisor.

Basándose en un concepto orientado a la ejecución de instrucciones de ensamblador y con la necesidad de derivar sus requisitos de virtualización, P&G introdujeron una clasificación de instrucciones de un ISA en 3 grupos diferentes:

- **Instrucciones privilegiadas**: aquellas a capturar si el procesador está en modo usuario y que no se capturan si está en modo privilegiado.
- **Instrucciones sensibles al control**: aquellas que pretenden cambiar la configuración de los recursos del sistema.
- **Instrucciones sensibles al comportamiento**: aquellas cuyo comportamiento o resultado dependen de la configuración de los recursos (contenido del registro de "relocation" o del modo del procesador).

Estos dos últimos conjuntos de instrucciones se agrupan como "instrucciones sensibles" y en términos de virtualización son prácticamente igual de importante que las instrucciones privilegiadas a pesar de que no generen un cambio de modo del procesador.

El resultado principal del análisis de P&G se puede expresar a través de los dos siguientes teoremas que resumen la formalización:

Teorema 1: para cualquier computadora convencional de tercera generación, el hipervisor de la máquina virtual(VMM) puede ser construido si el repertorio de instrucciones sensibles para esa computadora es un subrepertorio del repertorio de instrucciones privilegiadas.

Intuitivamente, el teorema dice que para construir un VMM, es suficiente que todas las instrucciones que pudiesen afectar al correcto funcionamiento del VMM (instrucción sensible) se capturen y pasen el control al VMM siempre. Esto garantiza el control de propiedad de los recursos. En cambio, las instrucciones no privilegiadas deben ejecutarse nativamente por eficiencia.

Un problema parecido es el de derivar los requerimientos de la ISA para virtualización recursiva, esto es, las condiciones por las cuales un VMM que puede ejecutar una copia de sí mismo se puede construir. En el siguiente teorema se exponen estos requerimientos.

Teorema 2: una computadora convencional de tercera generación es recursivamente virtualizable si:

1. Es virtualizable.
2. Un VMM sin dependencias temporales puede ser construido para ello.

A través de ambos teoremas se presentó la primera definición científica de cuáles eran los requisitos de forma y de fondo de una máquina virtual.

Paravirtualización

Pasamos ahora a hablar de la paravirtualización, que en un principio fue la primera opción que planteamos para llegar al modelo de aplicación que terminamos implementando.

Mediante el modelo de paravirtualización, la máquina virtual no corre sobre un host con hardware simulado, sino que se ofrece una API que necesita de la modificación del sistema operativo "guest" para realizar las llamadas a ella.

Surge para resolver el problema que presentan los repertorios en los que la instrucciones sensibles no son un subconjunto de las privilegiadas, como explicamos anteriormente.

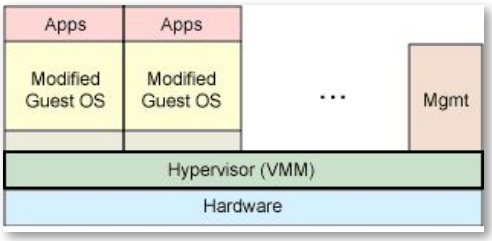


Ilustración 6: Paravirtualización

La implementación más conocida de paravirtualización es Xen.

Xen

Xen por medio de la paravirtualización alcanza un alto rendimiento; es decir, bajas penalizaciones del rendimiento, típicamente alrededor del 2%, con los peores casos de rendimiento rondando el 8%; esto contrasta con las soluciones de emulación que habitualmente sufren penalizaciones de un 20%.

A diferencia de las máquinas virtuales tradicionales, que proporcionan entornos basados en software para simular hardware, Xen requiere portar los sistemas operativos para adaptarse al API de Xen. Hasta el momento hay ports para NetBSD, Linux, FreeBSD y Plan 9.

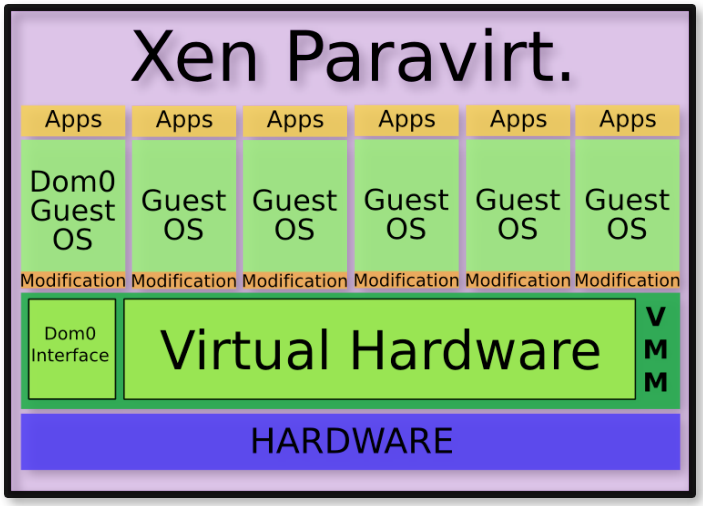


Ilustración 7: Paravirtualización: Xen

Virtualización a nivel de Sistema Operativo

La virtualización a nivel de sistema operativo suele referirse al modelo "sandbox", que consiste en aislar parte de los procesos del sistema operativo del resto del sistema y haciendo uso del propio kernel común para aumentar el rendimiento implementando así una máquina virtual. Suele girar en torno a la llamada "chroot".

Un *chroot* en un sistema operativo Unix es una operación que cambia el directorio raíz, afectando solamente al proceso actual y a sus procesos hijos. "chroot" se refiere a la llamada de sistema `chroot(2)` o al programa ejecutable `chroot(8)`.

El sistema chroot fue introducido por Bill Joy el 18 de marzo de 1982 (17 meses antes de que 4.2BSD fuera liberado) para probar su sistema de instalación y construcción. No fue pensado como un mecanismo de seguridad.

Un programa que se ejecuta en un entorno con el directorio raíz cambiado, no puede acceder a archivos fuera de ese directorio. A menudo es malentendido como un dispositivo de seguridad, usado en un vago intento de crear una zona segura para correr un programa que provoca desconfianza, o que no está probado o que de algún modo es peligroso, como si chroot fuera un mecanismo real de jaula funcional.

En la práctica, hacer un chroot es complicado ya que los programas esperan encontrar en lugares predeterminados el espacio de almacenamiento, archivos de configuración, sus bibliotecas de

enlace dinámico. Para habilitar los programas que puedan correr dentro de un directorio chroot, se debe incluir también un conjunto mínimo de estos archivos, ya que el programa no puede acceder fuera de la jaula.

Los programas tienen permitido llevarse descriptores de archivos abiertos (sean de archivos físicos, de tuberías, o de conexiones de red) dentro del chroot, lo cual puede simplificar el diseño de la jaula haciendo innecesario dejar archivos funcionales dentro del directorio chroot. Esto también funciona como un sistema de capacidades simple, en el cual, al programa se le otorga acceso explícito a los recursos externos del chroot basado en los descriptores que puede llevar a su interior.

Hablaremos de las implementaciones más conocidas de este modelo: FreeBSD jails y Linux Vserver.

FreeBSD jails

El mecanismo *FreeBSD jail* es una implementación de virtualización a nivel de sistema operativo que permite a los administradores particionar un sistema FreeBSD en diversos mini sistemas independientes llamados *jails* o *jaulas.*

La necesidad de crear FreeBSD jails vino del deseo de proveedores de servicios de establecer un corte separa torio entre sus servicios y sus consumidores, principalmente por razones de seguridad y facilidad administrativas. En lugar de añadir una nueva capa de opciones de configuración de grano fino, la solución tomada fue la de dividir por compartimentos el sistema, tanto sus recursos como sus ficheros, de tal manera que sólo la persona adecuada puede acceder al compartimiento correcto.

Linux VServer

Linux-VServer es un mecanismo de jaula que se puede usar para particionar con seguridad recursos en un sistema (tales como el sistema de ficheros, el tiempo de CPU, direcciones de red y memoria), de tal manera que los procesos no pueden montar ataques de "denegación de servicios" en nada que esté fuera de su partición.

A cada partición se le llama *contexto de seguridad*, y el sistema virtualizado en el que se encuentra es el *servidor privado virtual.* Los servidores virtuales se usan comúnmente en servicios web de almacenamiento, donde son útiles para segregar cuentas de clientes, reunión de recursos y contener cualquier violación de seguridad potencial.

Soluciones en plataforma ARM

Pasamos ahora a estudiar el estado del mercado de la virtualización para arquitecturas ARM. Hablaremos sobre las alternativas más conocidas de las que se disponen: VirtualLogix VLX y VMware MVP.

VirtualLogix VLX

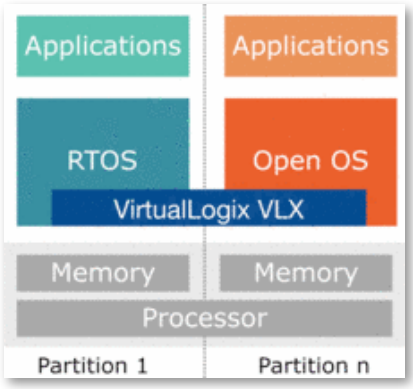


Ilustración 8: VLX

Integrando tecnología de virtualización en tiempo real, VLX para sistemas empotrados permite la ejecución simultánea de múltiples sistemas operativos en rendimiento crítico y en plataformas de procesadores mono y multinúcleo. Se implementa un particionado virtual de los recursos disponibles(memoria y CPU) y se asignan a cada SO. Así, cada SO podrá usar sus propios y políticas, como la gestión de memoria, sin interferir con otros sistemas operativos virtualizados.

Para poder asegurar la eficiencia, VirtualLogix emplea técnicas de paravirtualización, es decir, algunas adaptaciones del kernel del SO virtualizado las ha realizado VirtualLogix.

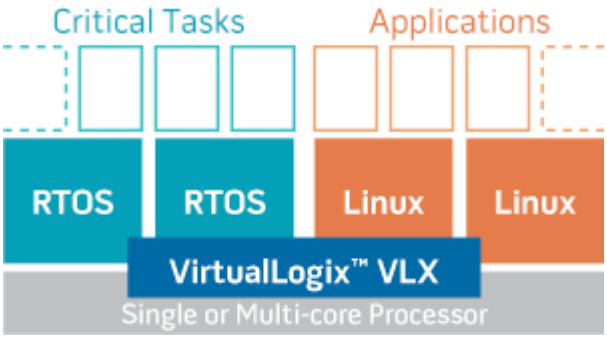


Ilustración 9: VLX para entornos embebidos

VMware MVP

Construido sobre tecnología adquirida de Trango V, VMware MVP es una capa de software que está empotrada en el teléfono móvil para disociar las aplicaciones y los datos del hardware

subyacente. Está optimizado para funcionar de manera eficiente en teléfonos móviles de bajo consumo energético y memoria restringida. MVP actualmente soporta una amplia gama de sistemas operativos en tiempo real tales como Windows CE 5.0 y 6.0. Linux 2.6.x, Symbian 9.x, eCos y μC/OS-II.

Alternativas de desarrollo descartadas

Microkernel + hipervisor

Cuando comenzamos a planificar lo que sería nuestra máquina virtual tomamos como punto de partida la aproximación realizada por VirtualLogix. Como se explico en apartados anteriores además de usar un hipervisor basado en un microkernel que gestiona también realizar particiones virtuales del hardware.

Nuestro primer acercamiento fue obviar la virtualización de la CPU, pero mantener la partición de la RAM. Como se puede ver en la siguiente figura nuestra propuesta quedó así:

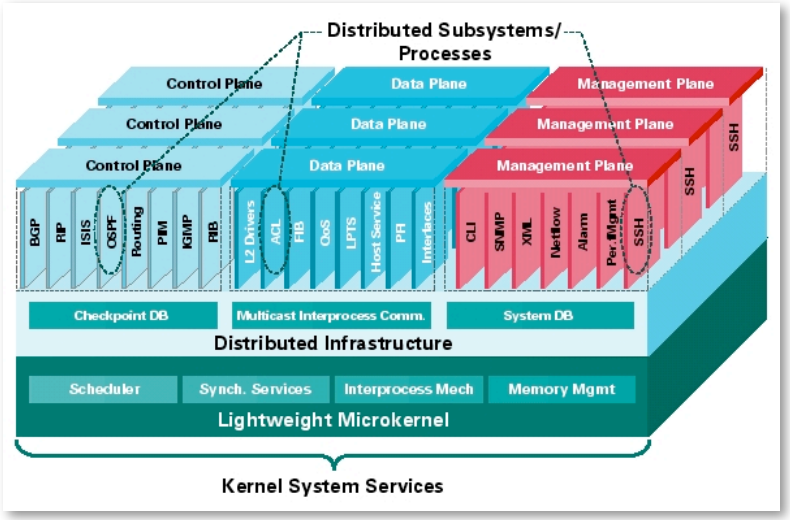


Ilustración 15: microkernel + hipervisor

Esta primera propuesta resultaba muy compleja. En primer lugar debíamos usar un micro-kernel, lo que nos obligaba a pasar por un período de aprendizaje de programación en estos sistemas. Además, debíamos pasar por la etapa de paravirtualización inexorablemente.

La etapa de paravirtualización, esto es, modificar un sistema operativo para poder usarlo con nuestro hipervisor fue el factor determinante que no hizo renunciar a esta propuesta, pues en el tiempo del que disponíamos nos resultaba imposible implementar el hipervisor basado microkernel y además modificar algún sistema operativo para poder usarlo.

Como base para el hipervisor valoramos usar tanto un microkernel (OKL4 era el mejor posicionado por tener licencia GPL y haber sido ya usado en distintos proyectos comerciales) como algún sistema operativo en tiempo real(eCos y FreeRTOS eran nuestras primeras opciones tanto por la extensa documentación existente para ambos como por su carácter OpenSource).

Hipervisor sobre un host linux

El siguiente paso en la planificación de nuestro proyecto fue considerar como solución la paravirtualización al estilo Xen, pero por supuesto a menor nivel y con unos objetivos mucho menos ambiciosos.

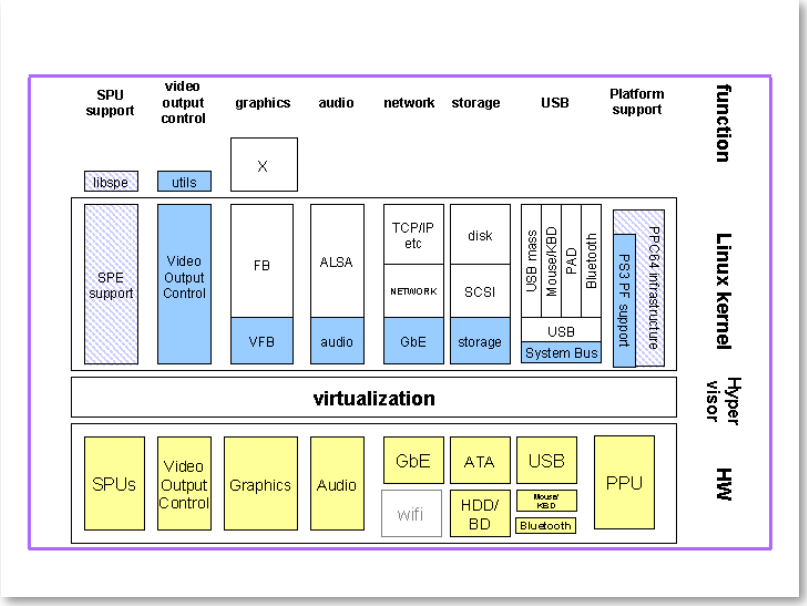


Ilustración 16: linux + hipervisor

Así, pasamos por plantear un demostrador en el que se modificara un sistema operativo Linux para permitir la ejecución de otro sistema operativo como "guest". Como ya se explico anteriormente se realizan cambios en la gestión de llamadas al sistema de tal manera que en vez de que el sistema operativo "guest" las ejecute, cuando una llamada al sistema es invocada se produce una interrupción del sistema operativo "host" que le permite tomar control de la situación y realizar diversas comprobaciones.

Para llevarlo a cabo había que realizar dos modificaciones:

1. En primer lugar, había que modificar la gestión de las "syscall's" para permitir que fuera nuestro hipervisor el que las ejecutara.

2. En segundo término, tendríamos que implementar en el sistema operativo "host" un nuevo anillo de permisos donde se ejecutara el sistema operativo "guest".

Una vez analizadas y valoradas, llegamos a la conclusión de que ambas modificaciones excedían los propósitos que un proyecto de Sistemas Informáticos debe tener. Si bien se puede argumentar que se podría haber realizado un demostrador tecnológico limitado, este demostrador incluía realizar las dos modificaciones ya comentadas, pues son la base sobre las que las llamadas al que habría que modificar también se realizarían. Por estos motivos, esta propuesta fue descartada.

Soluciones descartadas

Hemos investigado sobre distintas maneras de encriptar sistemas de ficheros para proporcionar una mayor protección a la máquina virtual a la hora de accesos desde fuera de nuestra jaula. A continuación vamos a describir las alternativas que hemos llegado a considerar.

Encriptación del sistema de ficheros: Truecrypt

Este software tiene una serie de características que enumeramos a continuación**.**

- Crea un disco virtual encriptado en un fichero y lo monta como si fuera un disco físico.
- Encripta particiones enteras o dispositivos como USB's o discos duros. Además, la encriptación es automática, en tiempo real ("onthefly") y transparente
- Permite encriptar una partición o dispositivo incluso con un sistema operativo Windows instalado para después poder acceder a él. Permite también volúmenes y sistemas operativos ocultos.
- Proporciona dos niveles de seguridad en caso de obtención de la contraseña por un extraño.
- Algoritmos de encriptación que soporta: AES-256, Serpent y Twofish.

Procedimos a instalarlo en un sistema operativo Ubuntu 8.10. En el repositorio se encuentra la versión 6.1.a0. Una vez instalado tenemos la siguiente interfaz que es muy intuitiva y fácil de utilizar:

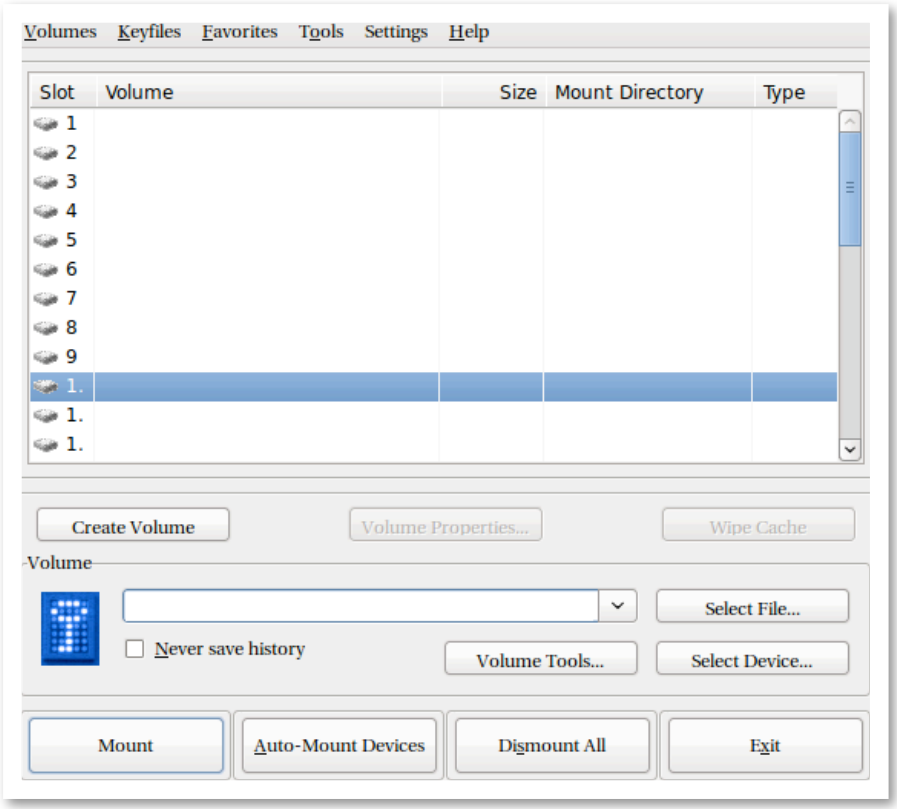


Ilustración 10: TrueCrypt (pantalla de configuración)

Aun así lo descartamos por la práctica imposibilidad de integrarlo con nuestro código. Aunque la valoración en todo caso ha sido muy positiva.

Encriptación del sistema de ficheros: Volúmenes cifrados

Sobre una máquina virtual Virtualbox hemos instalado Debian en su última versión "Lenny" con objeto de investigar sobre la encriptación de particiones enteras. Ésta es la estructura inicial con la que queríamos trabajar:

- Partición de 100MB cifrados montada en /*home*/*ghost* (aep 256bits)
- Partición primaria de unos 3GB montada en / (ext3)
- Partición de unos 700MB montada en /*home* (ext3)

Durante la instalación debemos realizar algunos ajustes:

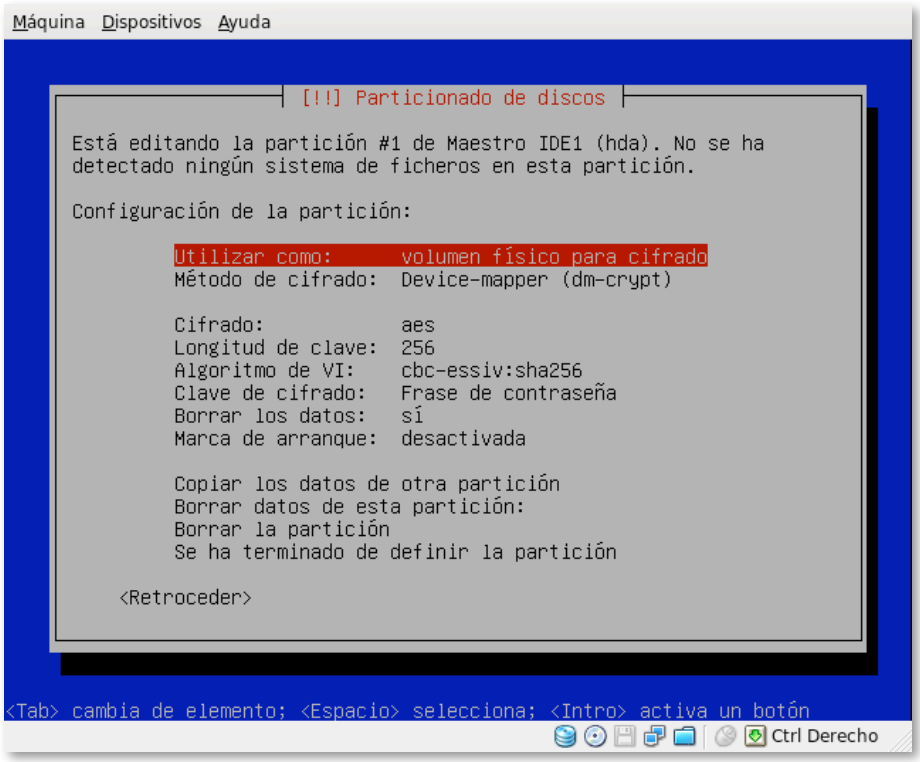


Ilustración 11: encriptación en Debian I

A la hora de seleccionar el tipo de partición (normalmente para UNIX usaremos extended-3 a no ser que requiramos un FS en concreto), que debe ser de tipo cifrado. Al finalizar el proceso de "Configurar partición" nos aparece una opción nueva en la herramienta de particionado: "configurar las volúmenes cifrados", como podemos ver en la siguiente captura:

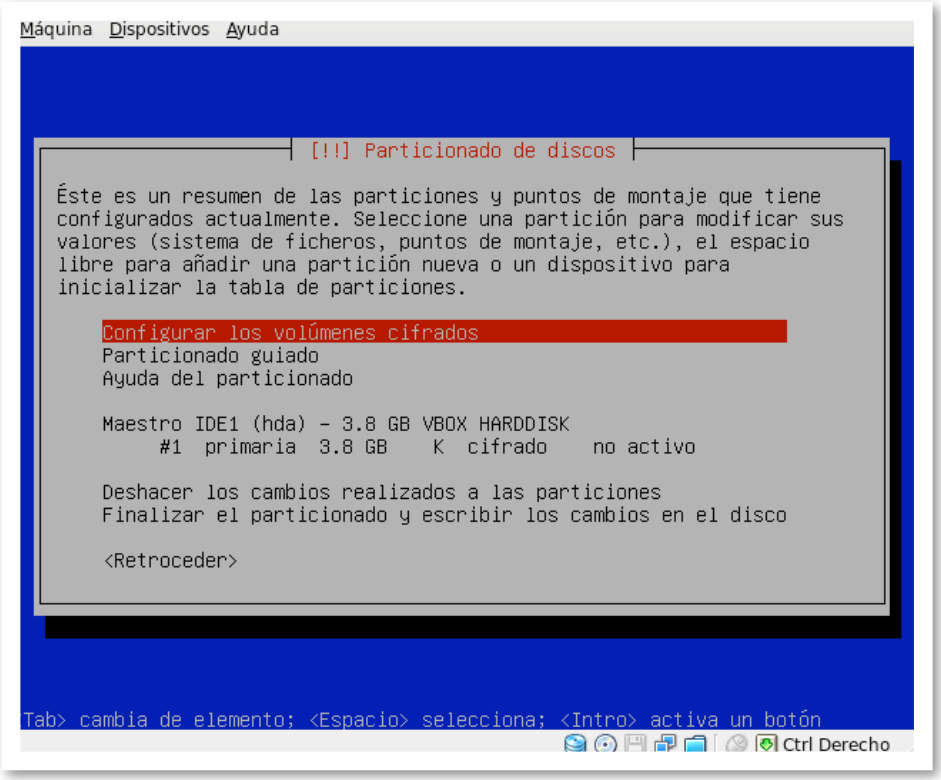


Ilustración 12: Encriptación en Debian II

Una vez seleccionada esta opción la configuración es muy simple nos pide el tipo de algoritmo a usar y la contraseña de encriptación. Finalizamos la instalación como habitualmente lo haríamos y ya disponemos de un volumen cifrado.

El principal inconveniente y motivo suficiente para descartarlo a la hora de incluirlo en nuestro proyecto es que en el arranque del sistema se pide la contraseña de cifrado del dispositivo y hasta que no se la proporcionemos el sistema no va a arrancar, ni aunque la partición sea no-critica. El segundo inconveniente es que una vez montado el volumen se maneja exactamente igual que una partición no cifrada por lo que la seguridad que le pretendíamos dar a nuestra máquina virtual no nos la podía proporcionar.

Encriptación del sistema de ficheros: Fuse

Encfs es un desarrollo que permite encriptar parte o la totalidad del sistema de ficheros, todo ello basado en Fuse. Utilizando esta tecnología podemos crear volúmenes encriptados con un simple comando.

Para ello se debe descargar el paquete “fuse” o “fuse-utils” depende de la distribución que estemos utilizando, además, deberemos instalar encfs:

```
sudo apt-get install encfs
```

```
sudo modprobe fuse
```

Si queremos que el modulo "fuse" arranque con el sistema tendremos que añadirlo en */etc/modules*.

Luego el usuario tiene que añadirse al grupo "fuse":

```
sudo adduser your_username fuse
sudo usermod -a -G fuse username
```

Esto último es para evitar este error:

```
fuse: failed to exec fusermount: Permission denied
fuse failed.  Common problems:
fuse kernel module not installed (modprobe fuse)
invalid options -- see usage message
```

Más tarde crearemos dos directorios, uno para los archivos y otro como punto de montaje:

```
mkdir ~/dir
mkdir ~/dir_mnt
```

Para crear el volumen encriptado no tenemos más que ejecutar el siguiente comando:

```
encfs ~/dir ~/dir_mnt
```

Tenemos que tener cuidado pues encfs utiliza siempre direcciones absolutas (mas tarde veremos que eso es un problema para nuestro propósito).

La primera vez que se ejecute el comando se procederá a dar toda la información relativa al volumen (tamaño de bloque, algoritmo, longitud de clave, clave...) Las siguientes solo se solicitara la clave para acceder. La idea es que podríamos trabajar en */dir* y en *dir_mnt* estaría encriptado. Para desmontar el volumen simplemente tendríamos que hacer:

```
fusermount -u ~/dir_mnt.
```

Hemos descartado también esta tecnología por el motivo anteriormente citado, al necesitar trabajar con rutas absolutas se produce un posible agujero de seguridad que un usuario malintencionado podría utilizar para acceder al resto del sistema.

Red virtual

Dado que una de las funcionalidades deseadas es tener acceso a redes (redes locales, impresoras en red, Internet) era lógico realizar investigaciones en esa dirección. En un principio investigamos sobre el posible uso de puentes e interfaces virtuales.

Un puente o bridge es un dispositivo de interconexión de redes de ordenadores que opera en la capa 2 (nivel de enlace de datos) del modelo OSI. Interconecta dos segmentos de red (o divide una

red en segmentos) haciendo el pasaje de datos de una red hacia otra, con base en la dirección física de destino de cada paquete. Un *bridge*, por tanto, conecta dos segmentos de red como una sola red usando el mismo protocolo de establecimiento de red.

Funciona a través de una tabla de direcciones MAC detectadas en cada segmento al que está conectado. Cuando detecta que un nodo de uno de los segmentos está intentando transmitir datos a otro, el bridge copia la trama hacia la otra subred. Por utilizar este mecanismo de aprendizaje automático, los puentes no necesitan configuración manual.

La principal diferencia entre un bridge y un *hub* es que el segundo pasa cualquier trama con cualquier destino para todos los otros nodos conectados, en cambio el primero sólo pasa las tramas pertenecientes a cada segmento. Esta característica mejora el rendimiento de las redes al disminuir el tráfico inútil.

Se distinguen dos tipos de bridge:

- **Locales**: sirven para enlazar directamente dos redes físicamente cercanas.
- **Remotos o de área extensa**: se conectan en parejas, enlazando dos o más redes locales, formando una red de área extensa, a través de líneas telefónicas.

En nuestro caso estamos interesados en los puentes a nivel local. Para manejarlos tenemos dos alternativas, modificar directamente el archivos */etc/network/interfaces* o utilizar el paquete bridge-utils que explicaremos ahora.

Al instalar el paquete bridge-utils tendremos accesible el comando "brctl" que sirve para configurar, mantener e inspeccionar la configuración de los puentes Ethernet en Linux. Por ejemplo podemos crear un puente simplemente ejecutando el siguiente comando:

```
# brctl addbr  <nombre>
```

Si quisiéramos mostrar todos los puentes creados ejecutaríamos:

```
# brctl show
```

A continuación nos resulta indispensable hablar de los interfaces TUN/TAP de Linux. Un TAP de red es un dispositivo que nos proporciona acceso a datos a través de la red de un ordenador, cuando hablamos de un TAP virtual (como es el caso), hablamos de un dispositivo Ethernet simulado que actúa sobre la misma capa que las tramas Ethernet. Un TUN de red, simula un dispositivo en la subcapa de red que opera al mismo nivel que los paquetes IP.

Normalmente, para realizar una red virtual necesitamos tanto un puente como un dispositivo TAP, la idea es bastante intuitiva, con el TAP vamos a realizar un "clon virtual" de nuestro interfaz de

red (eth0 por ejemplo). Aquí mostramos el resultado final de la configuración del puente y de la interfaz:

```
auto lo
iface lo inetloopback


auto tap0
iface tap0 inet manual

up ifconfig $IFACE 0.0.0.0 up
down ifconfig $IFACE down
tunctl_user jaula
user

auto br0

iface br0 inet static

address 192.168.0.2
netmask 255.255.255.0

gateway 192.168.0.1
bridge_ports eth0 tap0
bridge_maxwait 0
```

Como se observa al puente br0 hay que especificarle la interfaz TAP a la que se va a conectar.

A pesar de la utilidad de los puentes e interfaces virtuales para la creación de subredes dentro de un mismo equipo, descartamos utilizarlos pues nos creaban problemas de seguridad dentro de la jaula: para acceder a las interfaces de red necesitaríamos dar acceso a la jaula a los dispositivos(/*dev*) y podría ser utilizado por cualquier programa para violar la seguridad de nuestra máquina virtual.

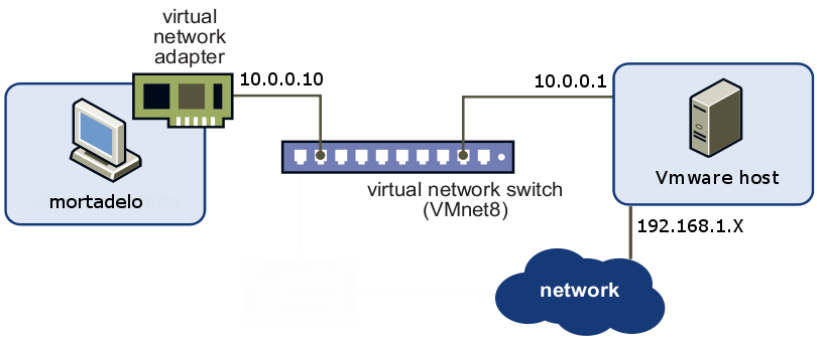


Ilustración 13: ejemplo de red virtual

Preparación del entorno

Con objeto de preparar un entorno de ejecución adecuado para nuestro demostrador se deben seguir algunas indicaciones para que el sistema quede en un estado adecuado. Explicaremos en detalle algunos pasos que se han de seguir. Para los demás referimos a la abundante documentación existente en Internet a este respecto.

Preparación de la jaula

Para la correcta ejecución de nuestro entorno "chroot" hay que seguir una serie de pasos

1. **Creación del directorio contenedor**: se debe crear una carpeta en el lugar que se elija más conveniente. Al realizar la llamada `chroot` será a esta carpeta a la que nos referiremos para arrancar nuestra máquina virtual. Más adelante daremos instrucciones para modificar nuestro código y fijar este directorio como punto de partida.

2. **Creación de las carpetas necesarias**: con el objetivo de establecer la máquina virtual como un contenedor capaz de ejecutar las aplicaciones que deseemos se deben crear las carpetas pertinentes dentro del directorio contenedor que hayamos elegido. Recomendamos la creación de los siguientes directorios:

   ```
   /bin
   /etc
   /home
   /lib
   /sbin
   /usr
   /usr/bin
   ```

3. **Copia de librerías, ejecutables y ficheros**: dentro de las carpetas creadas en el paso anterior se pueden copiar o enlazar los binarios y las librerías necesarias para su ejecución así como los ficheros que se crean conveniente (como ficheros de configuración). Para ver cuales son las librerías que necesita un ejecutable basta usar:

   ```
   ldd ejecutable
   ```

   Así mismo, para enlazar las librerías se puede usar:

   ```
   ln –s dirección_de_linkado objeto_a_linkar
   ```

   Recomendamos aplicar el principio del "mínimo imprescindible". Es decir, copiar o enlazar sólo aquello que se vaya a usar, nunca enlazar librerías "por precaución".

4. **Permisos**: por último, todos los ficheros copiados al directorio contenedor y las carpetas creadas dentro de él deben tener asignados los permisos mínimos necesarios para que el usuario que se encuentre en la jaula sea capaz de ejecutar las aplicaciones que el administrador crea pertinente que puede ejecutar.

Una vez finalizados estos pasos obtendremos un directorio que nuestra máquina virtual podrá usar como contenedor.

Compilación del Sistema Operativo

A partir de este momento es necesaria la compilación del kernel que suministramos con esta documentación. Este kernel tiene diversas modificaciones que se explicarán en secciones posteriores que permiten aislar el entorno de un conjunto de procesos del resto de procesos del sistema.

No ahondaremos en exceso en la configuración necesaria para correr el kernel ni en problemas derivados de una compilación incorrecta. Existe multitud de documentación al respecto. Las indicaciones básicas se detallan a continuación.

En primer lugar ejecutaremos:

```
make menuconfig
```

Y obtendremos un menú con el siguiente aspecto:

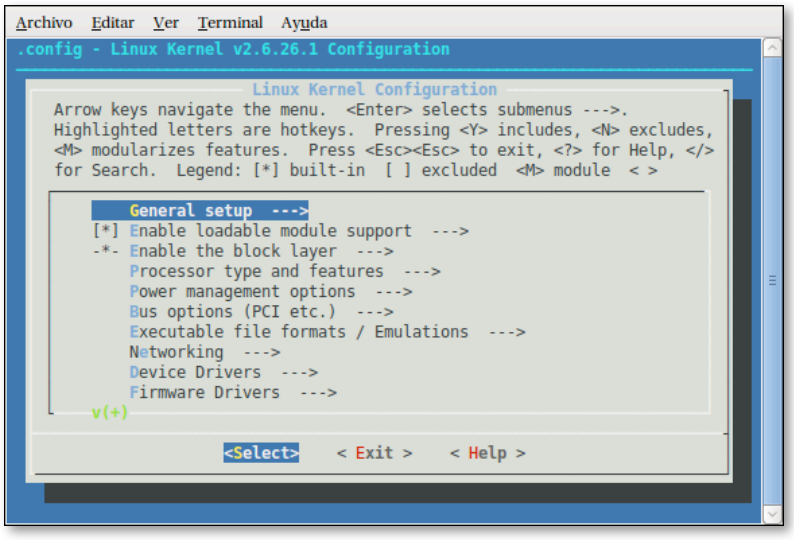


Ilustración 14: Configuración de los valores de compilación del kernel

Aquí seleccionaremos las opciones de compilación que deseemos. Para nuestra máquina virtual hemos usado la configuración por defecto en todos los apartados.

A continuación y con permisos de superusuario, ejecutaremos:

```
make bzImage modules modules_install install
```

Así, compilaremos el núcleo, los módulos e instalaremos ambos en sus directorios por defecto. Ya sólo nos queda un último paso:

```
cd /boot
mkinitramfs -o initrd-2.6.26.1 2.6.26.1
```

Ya hemos creado una imagen de carga completa.

Modificación del grub

El cargador de arranque grub también debe ser modificado. El menú del grub se encuentra en el fichero */boot/grub/menu.lst*. Es necesario agregar las líneas que permitan cargar el nuevo kernel pues éstas no se agregan automáticamente. Bastará con añadir las siguientes líneas al final:

```
#
title       Mi kernel
root        (hd0,1)
kernel      /boot/kernel-2.6.26.1/bzImage  root=/dev/sda2 ro
initrd      /boot/kernel-2.6.26.1.img
#
```

Quizá sea necesario cambiar alguno de los nombres aquí expuestos.

Ejecución del fichero jaula.c

Por último, sólo queda compilar y ejecutar el fichero jaula.c. En este fichero se encuentra el código que activa una jaula para el proceso que la ejecute y lanza una terminal desde donde poder interactuar. En la sección "Implementación" será explicada en detalle.

Para poder ejecutarla basta con compilarla:

```
gcc -o jaula jaula.c
```

Y ejecutarla con permisos de superusuario:

```
su root
./jaula
```

Con esto pasaremos a encontrarnos con una "Shell" ligeramente restringida y cuya capacidad dependerá de cómo lo haya configurado el Administrador.

Modelo de desarrollo

Tomando como modelos Linux-Vserver y FreeBSD Jail's, que son las dos implementaciones de referencia de virtualización a nivel de sistema operativo hemos planteado nuestra máquina virtual.

La elección de Linux se planteaba obvia, pues el código del kernel era fácilmente modificable y con gran soporte. Además, podíamos hacer uso de todas las herramientas de desarrollo y de la gran documentación disponible para este sistema operativo.

Después de un análisis minucioso, concluimos que las posibles modificaciones que podíamos asumir como "realizables" en el kernel eran las siguientes:

1. Creación de un entorno de ejecución aislado sin tener que recurrir a la paravirtualización.
2. Ejecución transparente por parte del usuario de aplicaciones en ese entorno.
3. Incorporación de nuevas llamadas al sistema que permitan la creación del entorno.
4. Incomunicación de los procesos internos del entorno de forma que resulte imposible comunicarse con el exterior.

El apartado número uno es sin duda el más general, pues engloba casi todos los demás. Sin embargo, aunque el concepto de paravirtualización se refiere únicamente a la modificación del "host" en un sistema virtual habitual, las modificaciones al sistema operativo resultan claras para conseguir los apartados dos y tres.

Hemos tenido, por tanto, que modificar el kernel de Linux para cumplir con los dos últimos apartados. La fase de investigación sugirió muchas posibilidades, pero han sido pocas las que finalmente hemos podido llevar a cabo.

A partir de ahora, al entorno de ejecución que simula una máquina virtual lo llamaremos "jaula" o "contenedor". El símil más factible es el de sandbox:

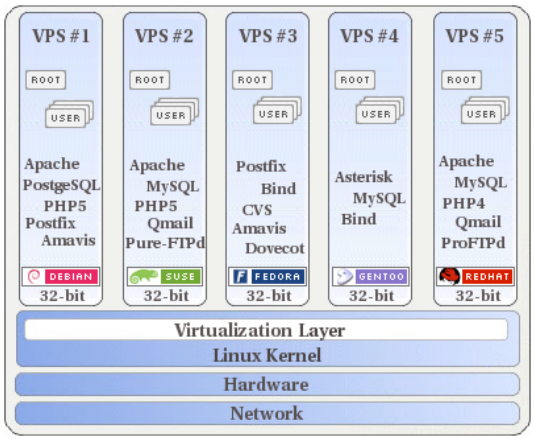


Ilustración 18: Jails en un sistema operativo

Dividiremos las tareas llevadas a cabo en dos aspectos: la modificación del sistema operativo y la aplicación implementada para ejecutar la máquina virtual sobre el kernel modificado. De las dos, la primera es la que ha requerido más esfuerzo.

Sistema operativo

Los cambios introducidos en el kernel tienen como el objetivo de que la aplicación del espacio de usuario ejecute de forma controlada. Distinguiendo los dos últimos apartados de la enumeración anterior:

1. **Inclusión de las nuevas llamadas al sistema**: estas llamadas tienen el objetivo de activar el mecanismo necesario para distinguir el proceso actual como una jaula.
2. **Incorporación de la jaula al propio kernel**: de esta manera, la jaula forma parte del propio núcleo, con lo que es más sencillo añadir características seguras y limitar su funcionamiento.
3. **Modificación del arranque**: con el objetivo de inicializar con éxito nuestra jaula.
4. **Aislamiento de los procesos(IPC's)**.

Y es en este apartado donde debemos detenernos. De las muchas posibilidades existentes(FIFO's, PIPE's, sockets, señales…) elegimos limitar el uso de los IPC("Inter Process Communication") por diversos motivos. El principal es que el código se encuentra en una carpeta independiente y está lo suficientemente separado del resto del kernel. Otros motivos serían la abundante documentación sobre IPC's o el hecho de que el código sea muy similar entre ellos.

El resto de opciones fueron descartadas, bien por considerarlas inviables, bien por no disponer del tiempo necesario. Algunos de estos descartes que fueron investigados más a fondo son explicados con algo de detalle en el apartado “Soluciones descartadas”.

Espacio de usuario

Finalmente, la aplicación del espacio de usuario realiza la llamada a “chroot” con diversos complementos que hemos añadido para que resulte más consistente(todos explicados en el apartado “Implementación”).

Implementación

Virtualización a nivel de sistema operativo

Realizados los descartes comentados en "Virtualización" se nos planteó la solución de forma obvia: realizar modificaciones menores en el sistema operativo. De esta manera acometimos la planificación de una propuesta mucho más cabal y dentro de nuestros propósitos y capacidades.

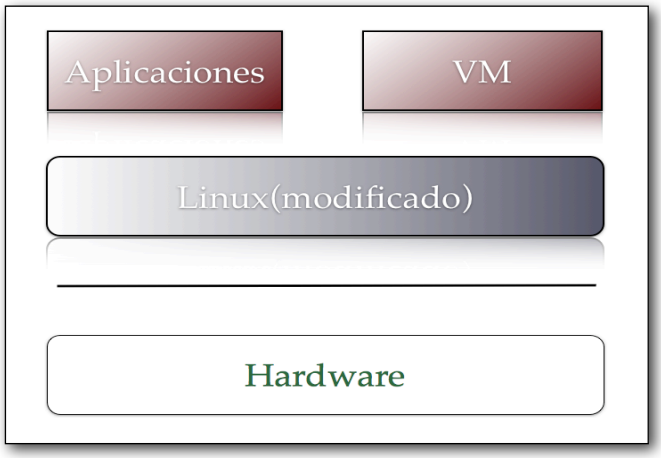


Ilustración 17: Linux con virtualización a nivel de sistema operativo

Este modelo se basa en la modificación del sistema operativo "host" para permitir la ejecución de "guest" de forma segura pero usando el propio kernel del "host". En apartados posteriores explicaremos con detalle la solución que hemos adoptado.

Introducción

La elección del modelo a desarrollar deja claros ciertos aspectos a implementar. Estos están explicados en "Modelo de desarrollo" y resulta evidente que lo primero a realizar es la inclusión de una serie de llamadas al sistema, en un kernel de Linux, que generen y obtengan la información de la jaula. Este paso obliga a la modificación de una estructura –*task_struct*– para poder generar una serie de contextos relacionados a las tareas de procesos, es decir, todo proceso tiene que pertenecer a un contexto determinado o, en nuestro caso, jaula. Por tanto, la información añadida a esta estructura será la que manejen las llamadas al sistema.

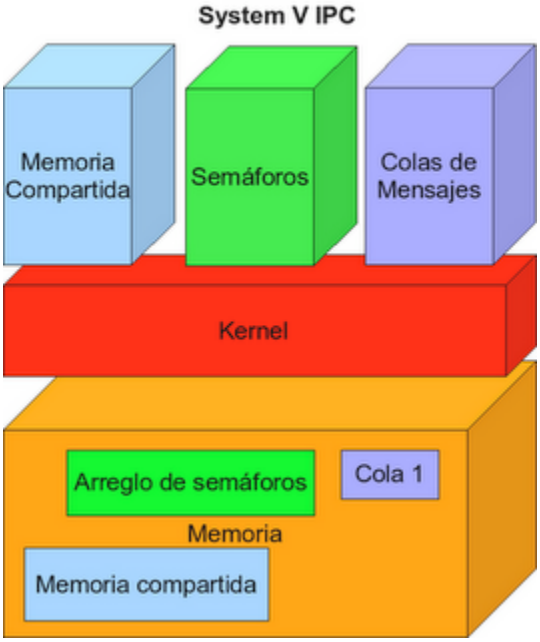


Ilustración 19: arquitectura de los IPC

Objetivos

El objetivo principal es poder diferenciar los grupos de procesos en contextos y que cada proceso pertenezca a una jaula. Es por ello que se necesita una continuidad de la información de la jaula, lo cual implica que otro objetivo sea la herencia de dicha información entre procesos cuando se encuentren en una misma jaula.

Además, se pretende que los procesos que se encuentren dentro de una jaula no puedan comunicarse mediante los mecanismos estándar con los procesos externos. Esto lo conseguiremos modificando algunos aspectos del sistema operativo.

Comentarios

A partir de ahora, en el caso de que sólo mostremos alguna parte en específico del código, marcaremos estas líneas con el número de línea en el que se encuentren en el fichero de código fuente original.

Las rutas que daremos se refieren siempre al código fuente del kernel. Así: */kernel/fork.c* se entenderá como la ruta relativa del fichero *fork.c* que se encuentra en la carpeta */kernel* del código fuente.

Contextos

Para la creación de contextos se ha elegido la modificación de la estructura *task_struct* que se encuentra en el fichero *sched.h*. Esta estructura es la que contiene la información básica de cada proceso y es por ello que el contexto al que pertenezca el proceso debe estar ahí colocado. La modificación de la estructura es sencilla y tan solo se ha añadido un número entero positivo para diferenciar jaulas o contextos:

```
1098: long jail;
```

La inicialización de *jail* se explica en el apartado siguiente.

Gracias a la modificación hecha ya se puede heredar el número de jaula cuando se crea un proceso. Sabemos que los procesos se crean en la llamada al sistema *fork()* copiando toda la información del proceso que lo crea o proceso padre. Esto significa que los momentos de ejecución que nos interesan para crear una jaula y que los procesos que hereden del proceso que la crea también pertenezcan a la misma jaula son la inicialización del sistema y la creación del nuevo proceso. Resulta necesario añadir código en dos sitios:

1. La función *fork_init()*ha de ser modificada para inicializar los jails del sistema:

   ```
   186: init_task.jail = 0;
   187: init_containers();
   ```

   De esta manera, el número de jaula del proceso *init* se marca a cero y el resto de procesos heredan así este número.

2. La función *copy_process()* también debe cambiarse de tal manera que cuando se copie la *task_struct* del proceso padre se incluya la copia del *jail*. La función *copy_process()* es llamada desde *do_fork()* para realizar el sistema "copy-on-write". El código introducido en *copy_process()* es:

   ```
   1095: p->jail = current->jail;
   ```

   Donde *p* es la *task_struct* del proceso nuevo que se crea y *current* se refiere al proceso que está realizando la llamada a *fork()*.

Llamadas al sistema

Las llamadas al sistema se han implementado tanto para plataforma x86 como para ARM. La principal diferencia es el punto de declaración en ensamblador(la diferencia radica en un fichero). Esto nos ha permitido realizar todos los cambios del kernel para ambas arquitecturas simultáneamente.

Desarrollo de create_newjail

Esta llamada al sistema es la encargada en crear una nueva jaula, es decir, que el proceso que quiere crear una nueva jaula pase a pertenecer a un contexto nuevo. A falta de mejoras, esto se consigue aumentando el número de jaula que tiene el proceso comprobando que no hay ningún proceso que ya tenga ese número de jaula.

Desarrollo de get_jailnumber

La otra llamada al sistema realizada es simplemente un acceso al número de jaula al que pertenezca el proceso. El código se puede reducir a:

```
return current->jail;
```

Inserción de las llamadas

Para crear las nuevas llamadas al sistema basta seguir el siguiente proceso:

1. Modificar el fichero */arch/x86/kernel/syscall_table_32.S* y añadir las siguientes líneas al final:

   ```
   329: .long sys_create_newjail
   330: .long sys_get_jailnumber
   ```

   Así se declaran las llamadas al sistema en la tabla de ensamblador que usará el enlazador para compilar el kernel.

   En el caso de tratarse de ARM el fichero a modificar en este caso sería */arch/arm/kernel/calls.S* con un código muy similar:

   ```
   368: CALL(sys_get_jailnumber)
   369: CALL(sys_destroy_jail)
   ```

2. Modificar el fichero */incluye/asm_x86/unistd_32.h*:

   ```
   335: #define __NR_create_newjail 327
   336: #define __NR_get_jailnumber 328
   ```

   Con estas líneas asignamos un número de llamada a cada una de nuestras nuevas llamadas. Este proceso es necesario porque cuando se realice la llamada al sistema y se produzca una interrupción software (“trap”) para salir del espacio de usuario y entrar en el espacio de ejecución del núcleo se usará este número para encontrar en la tabla la llamada al sistema que se ha producido.

   En el caso de que estemos declarando las llamadas para ARM, tendremos que usar el fichero */incluye/asm_arm/unistd.h*:

```
385: #define __NR_create_newjail (__NR_SYSCALL_BASE+355)
386: #define __NR_get_jailnumber (__NR_SYSCALL_BASE+356)
```

3. Finalmente también es necesario modificar el fichero/*include/linux/syscalls.h*:

   ```
   618: asmlinkage long sys_create_newjail(void);
   619: asmlinkage long sys_get_jailnumber(void);
   ```

   Mediante estas declaraciones dejamos constancia de que hay dos llamadas al sistema(además de las ya declaradas en el fichero) que deben compilarse y enlazarse.

   Este código también es apto para ARM.

Como vemos, la única diferencia entre arquitecturas se encuentran en dos ficheros donde se declaran las llamadas. Es un aspecto muy satisfactorio de la arquitectura interna de linux el separar de esta manera la implementación dependiente de la arquitectura de lo independiente.

La implementación ya descrita de las llamadas al sistema se ha situado en el fichero */jail/jail.c*. Todo el código que se implemente en el futuro puede hacer uso de este fichero y su cabecera para independizar siempre que se pueda la implementación sin modificar el sistema operativo.

Resultados de la inclusión de las llamadas al sistema

El resultado más importante es que se ha conseguido es que se creen contextos diferentes con las llamadas al sistema creadas y que tras las pruebas realizadas, esto lo hace correctamente tras poder comprobar que los números de jaula difieren. Otro resultado a señalar tras las pruebas es que la herencia de contextos funciona correctamente y que la pérdida de rendimiento es mínima al hacer una llamada a *fork()*.

IPC

Pasamos ahora a comentar cuáles han sido las modificaciones realizadas sobre la arquitectura de intercomunicación entre procesos, comúnmente conocida como IPC. Aunque existen otros mecanismos como Sockets, FIFO's, PIPE's, etc, nos hemos centrado en los IPC debido a que son los recursos más habitualmente utilizados.

Motivación de la modificación de los IPC

Tras haber realizado las llamadas al sistema y tener la jaula su propio contexto, es necesario el aislamiento de procesos lanzados desde el "interior" de la jaula de los procesos "externos" por motivos de seguridad. Por tanto, lo primero que se debía garantizar, en materia de seguridad, tras la llamada a *sys_create_newjail* es el aislamiento de los procesos sin pérdida de rendimiento. Como sabemos, la manera más eficiente de aislar procesos es controlando el uso de los recursos de

comunicación y sincronización entre procesos, esto es, limitar el acceso a los IPC's(semáforos, memoria compartida y mensajes) modificando su estructura y filtrando los IPC de manera que estos no puedan ser accedidos por procesos que no pertenezcan a su mismo contexto o jaula.

Objetivos de la modificación de los IPC

El objetivo principal del filtrado de uso de los IPC es ofrecer un alto grado de seguridad consiguiendo que los procesos ajenos a la jaula no puedan acceder a los IPC de los procesos del interior de la jaula y, sobre todo, los procesos del interior de la jaula no puedan obtener ni acceder a los IPC de los procesos ajenos.

El otro objetivo prioritario es conseguir que dicho filtrado no tenga una pérdida de rendimiento significativa y que por tanto las aplicaciones funcionen dentro de la jaula igual que lo harían fuera de ella cumpliendo así el principio de **equivalencia** de virtualización según Popek y Goldberg. Como detalle adicional y tal y como se ha mencionado anteriormente, hay que recordar que aquellos procesos que se ejecuten dentro de la jaula deben disponer de todo lo necesario para realizar su tarea en la carpeta sobre la que se realiza la llamada "chroot()".

Modificaciones necesarias

Las motivaciones y los objetivos ya comentados implican una serie de modificaciones necesarias para llevarlas a cabo. Una de estas modificaciones es el cambio de la estructura básica de los IPC para poder controlar en qué contexto se encuentran y así poder comparar con el de los procesos. El resto de modificaciones se encuentran en las llamadas al sistema correspondientes porque es donde se crean, modifican, manipulan y destruyen los IPC y es, por tanto, donde se ha de realizar el filtro de procesos.

Modificación de kern_ipc_perm

Dentro cada una de las estructuras de los diferentes IPC (*sem_arry, shmid_kernel* y *msg_queue*) existe un puntero a una estructura común, el *kern_ipc_perm* y es por ello que es el lugar elegido para añadir de un número que indique a qué contexto pertenece el IPC. La estructura presentará el siguiente aspecto:

```
/include/Linux/ipc.h
struct kern_ipc_perm
{
      spinlock_t        lock;
      int               deleted;
      int               id;
      key_t             key;
      uid_t             uid;
      gid_t             gid;
```

```
        uid_t           cuid;
        gid_t           cgid;
        mode_t              mode;
        unsigned long       seq;
        void            *security;
        unsigned long       jail;
};
```

Teniendo en esta estructura el contexto del correspondiente IPC, ya se puede comparar dicho contexto con el de los procesos obteniendo así la posibilidad de filtro deseado. Debido a que las comparaciones se realizan entre números enteros positivos la pérdida de rendimiento, a priori, es mínima y así obtener el segundo objetivo buscado.

Llamadas al sistema

Las llamadas al sistema que se deben modificar se pueden agrupar en:

1. Aquellas que obtienen el recurso IPC.
2. Las que crean el propio recurso.
3. Las que obtienen y/o modifican información acerca del recurso.
4. Las propias operaciones que ofrecen los IPC.

Se explicarán las modificaciones siguiendo la estructura de ésta agrupación.

Creación u obtención del IPC

Las llamadas al sistema que obtienen un nuevo IPC son *sys_semget*, *sys_shmget* y *sys_msgget* que están englobadas en la llamada *sys_ipcget*, que a su vez es donde comprueba si el recuso a devolver debe ser nuevo o uno ya publicado:

```
if (params->key == IPC_PRIVATE)
      return ipcget_new(ns, ids, ops, params);
else
      return ipcget_public(ns, ids, ops, params);
```

En el método *ipcget_new* hace una llamada al método correspondiente de cada recurso para la creación de un nuevo IPC. Estos métodos también son llamados por *ipcget_public* si el recurso a devolver no existe o no se encuentra y, por tanto, es en estos métodos donde se tiene que inicializar el número *jail* del IPC:

- Semáforos: el método *newary* crea el nuevo *sem_arry* y en el mismo lugar donde se inicializa el puntero *sem_perm* se ha colocado la instrucción de inicialización:

```
sma->sem_perm.jail = current->jail;.
```

- Memoria compartida: el método a modificar es *newseg* y de la misma manera que en los semáforos se ha añadido:

```
shp->shm_perm.jail = current->jail;
```

- Mensajes: el método *newque* crea el nuevo *msg_queue*:

```
msq->q_perm.jail = current->jail;
```

Por otro lado, el método *ipcget_public* también puede retornar un IPC ya existente y por tanto deben ser filtrados aquellos procesos que intenten obtener un objeto que no pertenezca a su contexto. Para ello se ha colocado, después de que se hagan las comprobaciones de seguridad rutinarias, la siguiente comparación:

```
if (ipcp.jail != current->jail)
      err = -34;
```

Con este filtro hemos conseguido que un proceso no pueda acceder a los IPC si no se encuentran en su misma jaula.

Modificación de información del IPC

Estas llamadas se denominan también llamadas de control. Estas llamadas tienen un comportamiento diferente dependiendo del comando a realizar, por lo tanto hay que diferenciar cuando es necesario filtrar y cuando no:

- Semáforos: se ha modificado la función *semctl_nolock* para los casos IPC_STAT y SEM_STAT; la función *semctl_main* para los comandos GETALL, GETVAL, GETPID, GETNCNT, GETZCNT, SETVAL y SETALL; y la función *semctl_down* para IPC_RMID e IPC_SET. En todos los casos se ha utilizado el código siguiente:

```
if(sma->sem_perm.jail != current->jail) {
      printk("Los números de jail difieren. \n");
      err = -34;
      goto out_unlock;
}
```

- Memoria compartida: se ha modificado la propia llamada al sistema *sys_shmctl* para los casos SHM_STAT, IPC_STAT, SHM_LOCK y SHM_UNLOCK; y la función *shmctl_down* para IPC_RMID e IPC_SET. El código de filtro para todos los casos es el siguiente:

```
if (shp->shm_perm.jail != current->jail) {
    err = -34;
    goto out_unlock;
```

```
}
```

- Mensajes: también se ha modificado la propia llamada al sistema *sys_msgctl* para los casos MSG_STAT e IPC_STAT; y como en los otros dos recursos la función *msgctl_down* para IPC_RMID e IPC_SET. El código de filtro para todos los casos es el siguiente:

```
if(msq->q_perm.jail != current->jail) {
      printk("Los números de jail difieren. \n");
      err = -34;
      goto out_unlock;
}
```

En todos estos casos se realiza una acción de dos posibles: muestra información del IPC privilegiada exclusiva al contexto al que pertenece o bien modifica dicha información pudiendo incluso cambiar el número de jaula del IPC. Con respecto al código utilizado cabe señalar dos aspectos importantes: el primero es que el código de error utilizado es completamente estocástico por lo que faltaría comprobar que no está siendo usado en ningún otro lugar del kernel; la etiqueta elegida para la instrucción *goto* (*out_unlock*) se debe a que los IPC se obtienen mediante un lock y por tanto debe de ser liberado. Éste código se encuentra ya implementado; nosotros lo hemos reutilizado.

Operaciones de los IPC

Las operaciones de los IPC son aquellas que hacen que interactúen los procesos entre sí. Las llamadas al sistema que implementan estas operaciones son:

- Semáforos: sólo tienen una llamada al sistema que implemente sus operaciones, *sys_semop*, que a su vez llama a otra, *sys_semtimedop*, siendo ésta la que tiene que llevar el filtro idéntico al de *sys_semctl*.
- Memoria compartida: aquí se encuentran la llamada *sys_shmat* que asigna una región de memoria a un proceso teniendo así que modificar el método *do_shmat* de la misma manera que en *sys_shmctl.* También está la llamada *sys_shmdt* que es una excepción al filtrado ya que no hace uso del recurso *shmid_kernel* y por tanto no hay que filtrar el método.
- Mensajes: los mensajes pueden realizar dos operaciones, enviar o recibir. Las llamadas al sistema que las implementan son *sys_msgsnd* y *sys_msgrcv* y los métodos que usan el recurso y por tanto a modificar son *do_msgsnd* y *do_msgrcv* de la misma manera, también, que en *sys_msgctl*.

Resultados

Los resultados obtenidos en las pruebas realizadas son altamente satisfactorios al conseguir que ningún proceso externo a la jaula obtenga un IPC del interior de la jaula y viceversa. Además, nuestros benchmarks han demostrado que la pérdida de rendimiento ha sido mínima.

Espacio de Usuario

En este punto, ya es posible crear una jaula y realizar el chroot para asignar el directorio raíz a dicha jaula. Por ello se ha desarrollado un código que genera de la jaula, realiza el chroot, actualiza los permisos adecuadamente para después lanzar el terminal correctamente. Este programa también está codificado en C lo que facilita el uso de las llamadas al sistema de Linux y las funciones del estándar POSIX.

Objetivos

El principal objetivo es tener una aplicación que al lanzarse cree una jaula y adecue el espacio de usuario al correcto funcionamiento de ésta. Esta aplicación debe comprobar que la jaula está bien generada y que no se han producido conflictos al realizar el **chroot**.

Conseguidos estos dos aspectos, la aplicación debe lanzar un terminal para que el usuario pueda lanzar otras aplicaciones y los comandos permitidos.

Desarrollo

Lo primero a realizar en esta aplicación es la creación de la jaula, por ello se llama a *create_newjail()*, comprobando que se cambie de contenedor correctamente. El siguiente paso importante a realizar es el **chroot**.

En primer lugar se debe hacer un cambio de directorio explícito:

```
if (chdir("dir")) {
        printf("Error al cambiar de directorio.\n");
        exit(-1);
    }
```

y cerrar todos los ficheros que estén abiertos en el momento de ejecución:

```
for (i = getdtablesize(); i > 3; i--) {
        while (close(i) != 0 && errno == EINTR);
}
```

Así pues, ya está el entorno preparado para llamar a **chroot**:

```
if (chroot("dir")) {
        printf("Error al crear la jaula.\n");
        exit(-1);}
```

Nótese que el directorio de argumento en **chdir** es necesariamente igual al de **chroot**, pues si no fuera así, habría problemas de acceso a archivos indicados con relaciones relativas.

Tras la ejecución del **chroot** hay que desactivar los permisos tanto del grupo como del usuario y, tras una serie de comprobaciones, adecuar las variables de entorno del usuario al de la jaula para poder lanzar la terminal de usuario.

```
setgid(1000);
setuid(1000);

char test[1024];
getcwd(test, 1024);
printf("Path actual: %s\n, test);
pw = getpwuid(getuid());

if(¡pw)
      exit(-1);

if (clearenv() < 0) {
printf("No se pudo limpiar el entorno.\n");
}
if (setenv("HOME", home, 1) < 0)
      printf("Error al configurar el directorio home\n");
if (setenv("USER", pw->pw_name, 1) < 0)
      printf("Error al configurar el usuario.\n");
if (setenv("SHELL", shell, 1) < 0)
      printf("Error al configurar el shell por defecto. \n");
if (setenv("USERNAME", pw->pw_name, 1) < 0)
      printf("Error al configurar el usuario. \n");
if (setenv("PATH", "/bin:/usr/bin:/usr/bin/sudo", 1) < 0)
      printf("Error al configurar el path. \n");
```

Finalmente, debemos ejecutar la terminal con la nueva ruta existente dentro de la jaula:

```
exec("/bin/bash", "/bin/bash", NULL);
```

Resultados

Ejecutando el fichero jaula.c obtenemos finalmente la máquina virtual funcional. Se crea una jaula con su propio número que la diferencia del resto del sistema y el mecanismo de seguridad de los ipc que hemos implementado entra en funcionamiento, impidiendo la comunicación con otros procesos que no pertenezcan a la jaula.

Líneas de trabajo futuras

A lo largo del desarrollo de nuestra máquina virtual hemos tenido ciertas ideas que o bien hemos descartado o hemos aparcado. Algunas de ellas porque necesitaban la base que hemos creado y otras porque hemos carecido de tiempo para implementarlas.

Al contrario que aquellas que mencionamos en el apartado "Soluciones descartadas", las que aquí mencionamos no llegaron a pasar a fase de desarrollo y se quedaron en la fase de diseño o fueron apareciendo a lo largo del desarrollo del proyecto de forma natural frente a aquello que en ese momento tratábamos.

1. **Tratamiento de errores**: sería conveniente que los errores de creación, destrucción, etcétera, de la máquina virtual o aquellos relacionados con la obtención de recursos para jaula tuvieran un reflejo en *perror()*, es decir, la función de la librería de C que devuelve el tipo de error a partir del código de error. Para conseguirla habría que modificar esta función e introducir nuestros códigos de error en ella.

2. **Mayor integración con el sistema de ficheros**: para que nuestra jaula resultara más trasparente al usuario sería conveniente que parte del sistema de ficheros, como por ejemplo */proc* o */dev* estuviera enlazadas dentro de la jaula de forma segura, es decir, sin poner en peligro la integridad del sistema **chroot**.

3. **Preparación de la jaula de manera automática**: se podría generar un script que generara la estructura de la jaula y copiara o enlazara los ficheros con los permisos adecuados de forma automática.

4. **Borrar una jaula desde la jaula cero**.

5. **Limitación de recursos**: otra solución factible es limitar los recursos asignados a una jaula (memoria RAM, ancho de banda, uso del procesador…) de forma que se pueda gestionar la carga que una jaula imprime al sistema. Esto es muy útil en el caso de querer ejecutar aplicaciones de servidor donde en cada jaula se ejecuta un programa de respuesta para la aplicación cliente y no deseamos que ninguna de ellas pueda saturar los recursos de la máquina.

6. **Mayor integración con el administrador**: una posibilidad extremadamente útil en caso de que la máquina virtual se pensara ofrecer de forma comercial sería desarrollar una serie de herramientas que permitieran al administrador hacer todos los apartados anteriores siendo capaz de seleccionar los parámetros que deseamos asignar a cada contenedor.

7. **Destrucción de los procesos de una jaula**: al salir de una jaula sería conveniente que todos los procesos que han quedado pendientes en ella y no cuelguen del proceso que creó la jaula sean terminados.

8. **Lista de procesos**: esta es una de las opciones más difíciles pero que garantizan la gran capacidad de desarrollo que tiene por delante nuestro proyecto.

   Habitualmente, tenemos una sola lista de procesos que ejecutan en un solo planificador. En caso de contar con varios procesadores, la lista de procesos se convierte en un híbrido procesos – tareas con capacidad de lanzar procesos a cada procesador. Si los procesos que ejecuten en una máquina virtual se aíslan en una lista de procesos independiente, se puede crear un planificador que gestione esa lista de procesos, de forma que no se mezclen con el resto de procesos del sistema y se puedan ejecutar en un procesador mientras el resto de procesos del sistema se ejecutan en otro.

   FreeVPS usa un mecanismo muy similar a éste que exponemos. Además, selecciona mediante un algoritmo estocástico (que tiene como variables la carga de cada lista de procesos y el tiempo que lleva ejecutándose) el planificador que pasa a estar activo en un procesador.

9. **Estructura propia**: la creación de una estructura propia para cada jaula también resulta indispensable. Sería en esa estructura donde se almacenaría la información sobre los recursos asignados a la jaula, las capacidades con las que comienza a ejecutar, diversa información estadística, direcciones IPv4 e IPv6, direcciones de acceso internas, ficheros abiertos, si la jaula se encuentra encriptada y el protocolo de desencriptación que habría que seguir… Esta estructura dotaría de un dinamismo a nuestro sistema del que actualmente carece.

Otras muchas se podrían seguir, pero hemos expuesto aquí las que consideramos más importantes para enfocar el desarrollo en tres sentidos:

- Aumento de las capacidades.
- Claridad.
- Facilidad de gestión.

Análisis

Introducción

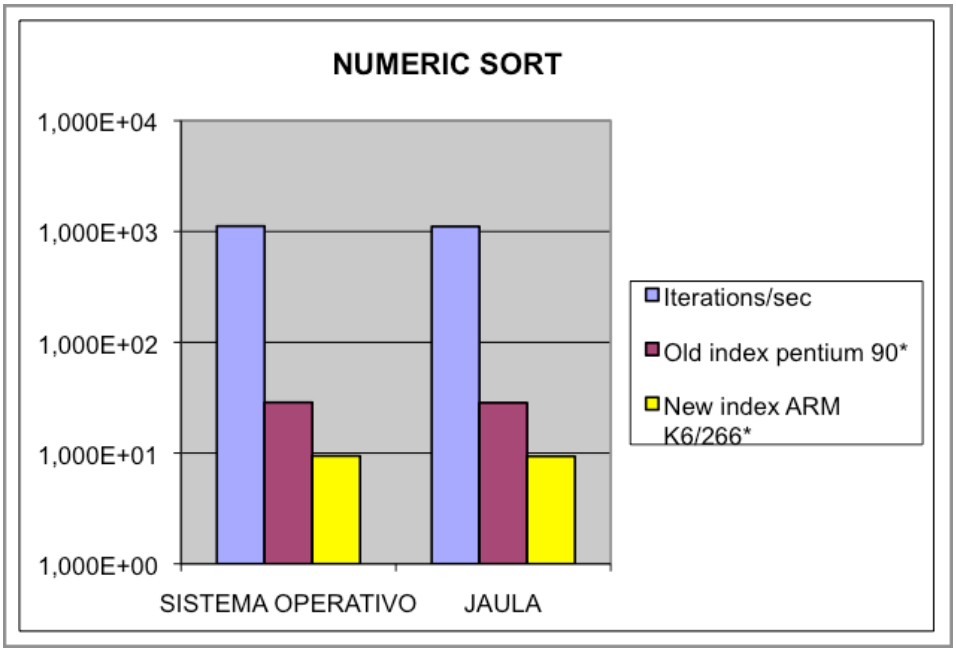
Todo proyecto necesita un análisis del resultado obtenido y por ello en esta sección vamos a encontrar los análisis de las características más importantes del proyecto entre las que se encuentran la implementación, las pruebas realizadas, las dificultades encontradas a lo largo del proyecto y las limitaciones del modelo y del sistema.

Bancos de pruebas

A continuación vamos a presentar los resultados obtenidos al realizar pruebas de rendimiento con los bancos de pruebas Nbench e Iozone. Se han realizado tres mediciones y hemos hecho la media aritmética de las tres. Los resultados no presentan resultados especialmente distanciados (< 1% diferencia) por lo que no hemos considerado necesario utilizar la media geométrica que es la que utilizan otros conjuntos de programas de pruebas como pueden ser los SPEC´s. Seguidamente vamos a realizar una breve descripción de los test y a comentar los resultados obtenidos.

Nbench

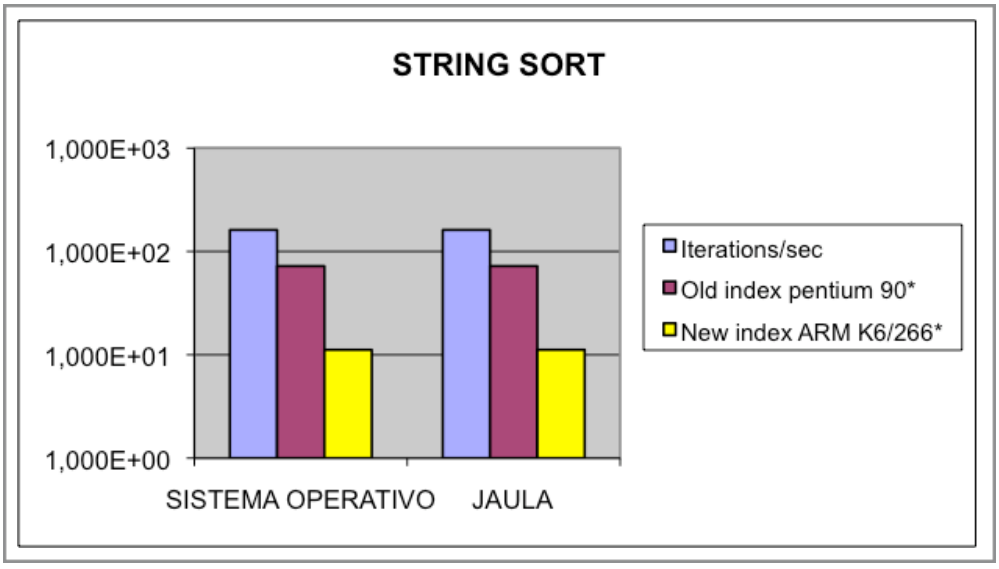
Este conjunto de pruebas se desarrolla sobre distintos ámbitos del sistema (operaciones con enteros, operaciones con cadenas, emulación de punto flotante, cálculos con series de Fourier, asignaciones y algunos otros). Dado que los resultados son tan parejos, adjuntar solamente las gráficas no nos sería de gran ayuda para analizar los resultados ya que las diferencias son mínimas, como primer ejemplo pondremos la gráfica de ordenación de números:



En cambio presentando los resultados en forma de tabla, podemos realmente observar las diferencias:

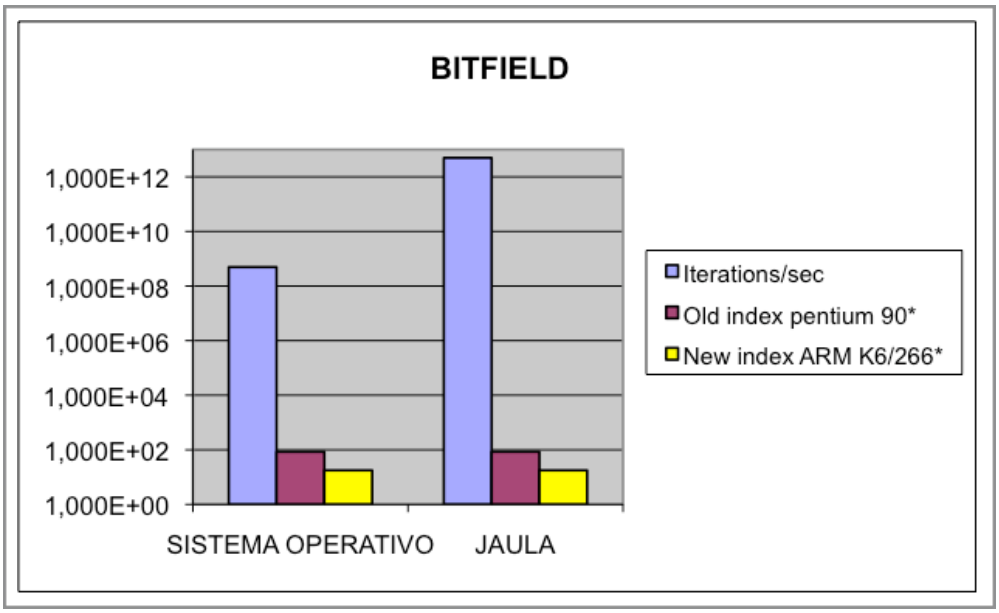
Numeric Sort		Iterations/sec	Old index pentium 90*	New index ARM K6/266*
	1st	1,112E+03	2,851E+01	9,360E+00
	2nd	1,124E+03	2,882E+01	9,470E+00
	3rd	1,117E+03	2,865E+01	9,410E+00
	Promedio	1,117E+03	2,866E+01	9,413E+00
Numeric Sort JAULA				
	1st	1,105E+03	2,834E+01	9,310E+00
	2nd	1,116E+03	2,862E+01	9,400E+00
	3rd	1,106E+03	2,835E+01	9,310E+00
	Promedio	1,109E+03	2,844E+01	9,340E+00

Como podemos comprobar la ordenación de números incluso es más eficiente dentro de la jaula que fuera de ella. De todas formas, el incremento de rendimiento es insignificante.



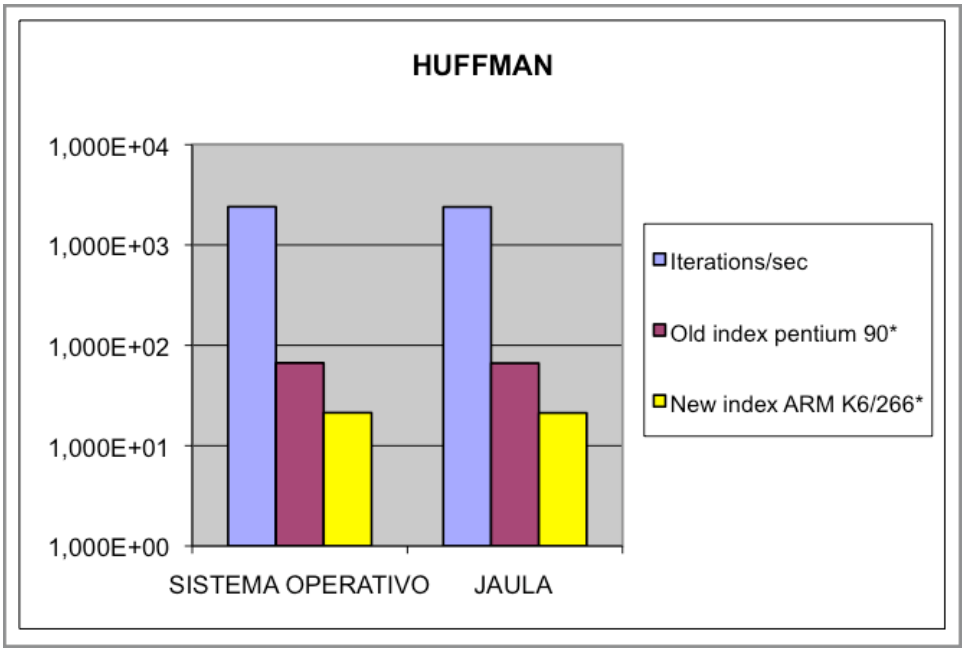
STRING SORT		Iterations/sec	Old index pentium 90*	New index ARM K6/266*
	1st	1,616E+02	7,219E+01	1,117E+01
	2nd	1,619E+02	7,232E+01	1,119E+01
	3rd	1,612E+02	7,201E+01	1,115E+01
	Promedio	1,615E+02	7,217E+01	1,117E+01
STRING SORT JAULA				
	1st	1,617E+02	7,225E+01	1,118E+01
	2nd	1,615E+02	7,217E+01	1,117E+01
	3rd	1,616E+02	7,221E+01	1,118E+01
	Promedio	1,616E+02	7,221E+01	1,118E+01

La ordenación de cadenas sí que sufre degradación en su rendimiento pero es mínimo.



BITFIELD		Iterations/sec	Old index pentium 90*	New index ARM K6/266*
	1st	4,945E+08	8,483E+01	1,772E+01
	2nd	4,948E+08	8,487E+01	1,773E+01
	3rd	4,936E+08	8,467E+01	1,769E+01
	Promedio	4,943E+08	8,479E+01	1,771E+01
BITFIELD JAULA				
	1st	4,940E+12	8,474E+01	1,770E+01
	2nd	4,942E+12	8,477E+01	1,771E+01
	3rd	4,949E+12	8,490E+01	1,773E+01
	Promedio	4,944E+12	8,480E+01	1,771E+01

Éste es el primer test que claramente resulta favorecedor para una de las alternativas, el numero de iteraciones que somos capaces de ejecutar dentro de la jaula es de un orden cuatro veces superior.



HUFFMAN		Iterations/sec	Old index pentium 90*	New index ARM K6/266*
	1st	2,403E+03	6,664E+01	2,128E+01
	2nd	2,405E+03	6,668E+01	2,129E+01
	3rd	2,409E+03	6,681E+01	2,134E+01
	Promedio	2,406E+03	6,671E+01	2,130E+01
HUFFMAN JAULA				
	1st	2,397E+03	6,647E+01	2,123E+01
	2nd	2,380E+03	6,600E+01	2,108E+01
	3rd	2,402E+03	6,661E+01	2,127E+01
	Promedio	2,393E+03	6,636E+01	2,119E+01

La última prueba también es muy similar aunque se puede apreciar una ligera pérdida de rendimiento observando el promedio de los tres campos.

Conclusión del banco de pruebas Nbench

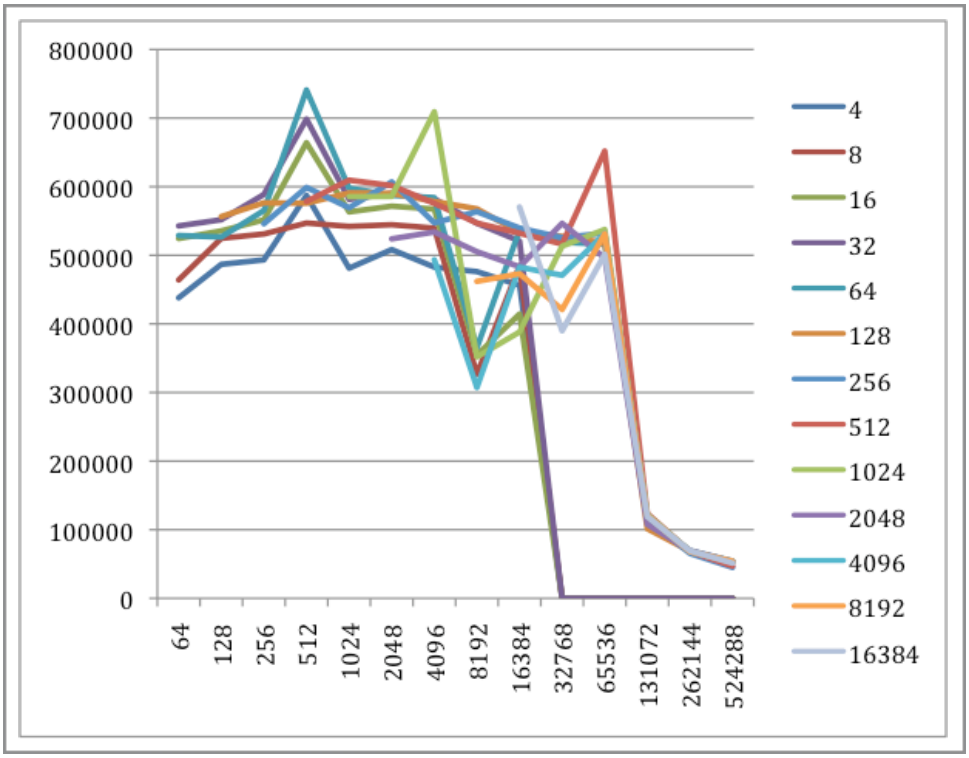
Excepto en la prueba bitfield (es un tipo de prueba que almacena flags sobre booleanos en un formato empaquetado, sería una equivalencia al formato horizontal de una microinstrucción) en el que dentro de la jaula podemos ejecutar 10.000 veces más instrucciones que fuera de ella, se observa que la perdida de rendimiento no llega ni al 1% con lo que podemos considerar un éxito moderado estos resultados teniendo en cuenta de que no está implementado al 100% todos y cada uno de los mecanismos de seguridad y funcionalidades de una máquina virtual completa, con lo que sería de esperar una disminución más acusada del rendimiento.

Iozone

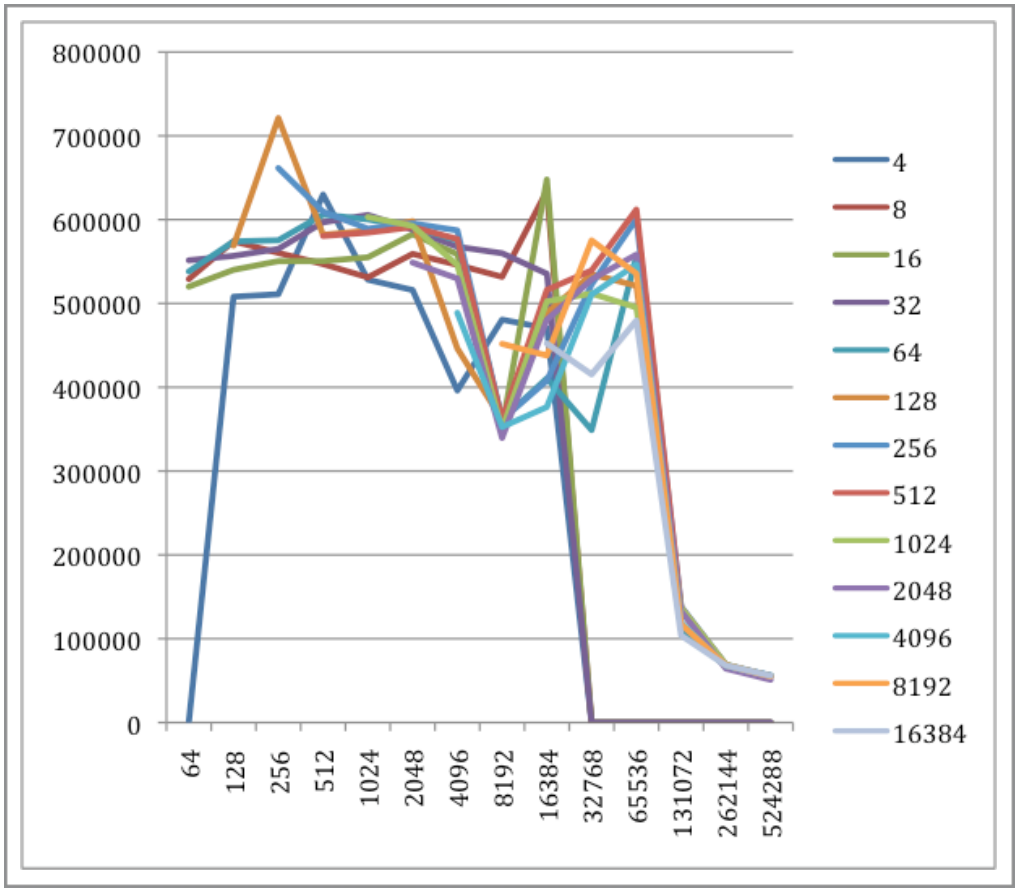
Ejecuta un conjunto de pruebas sobre las distintas modalidades de lectura y escritura de ficheros discriminando los resultados por tamaño de registro y por tamaño de fichero obteniendo el número de operaciones realizadas en cada una.

Resultados de escritura:

Sistema operativo



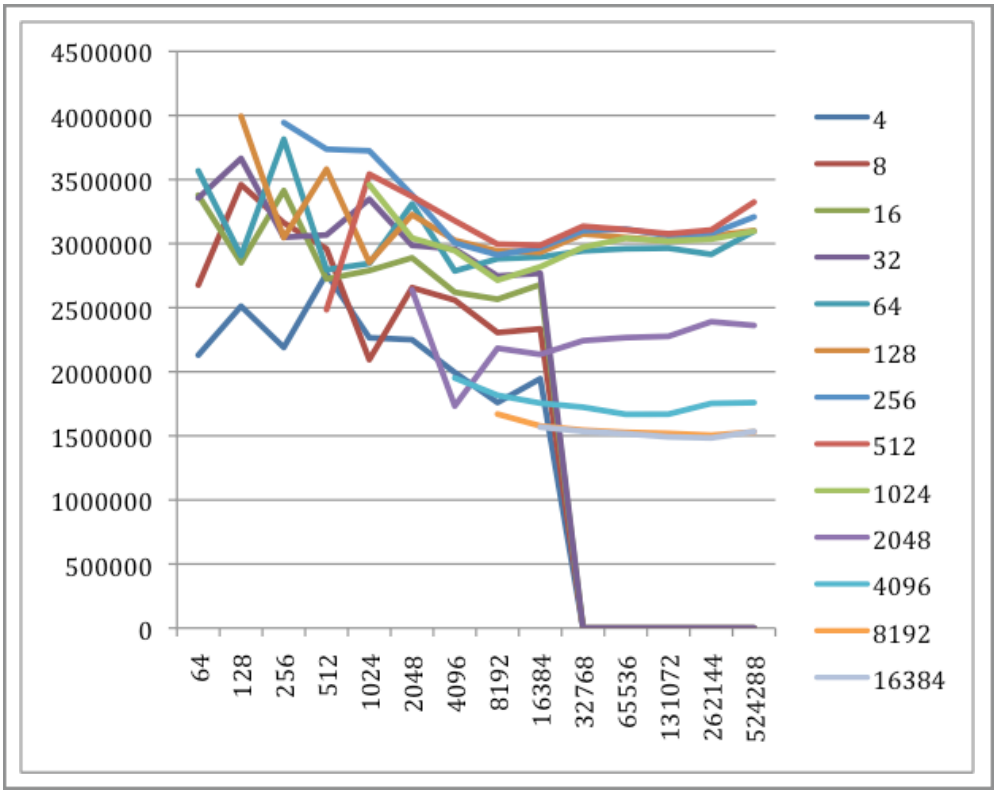
Jaula



Se puede observar en esta primera prueba que los resultados en ambos casos son similares lo que implica que la pérdida de rendimiento sea mínima. Esta pérdida es apreciable observando las tablas y no las gráficas que son realmente similares.

Resultados de lectura:

Sistema operativo

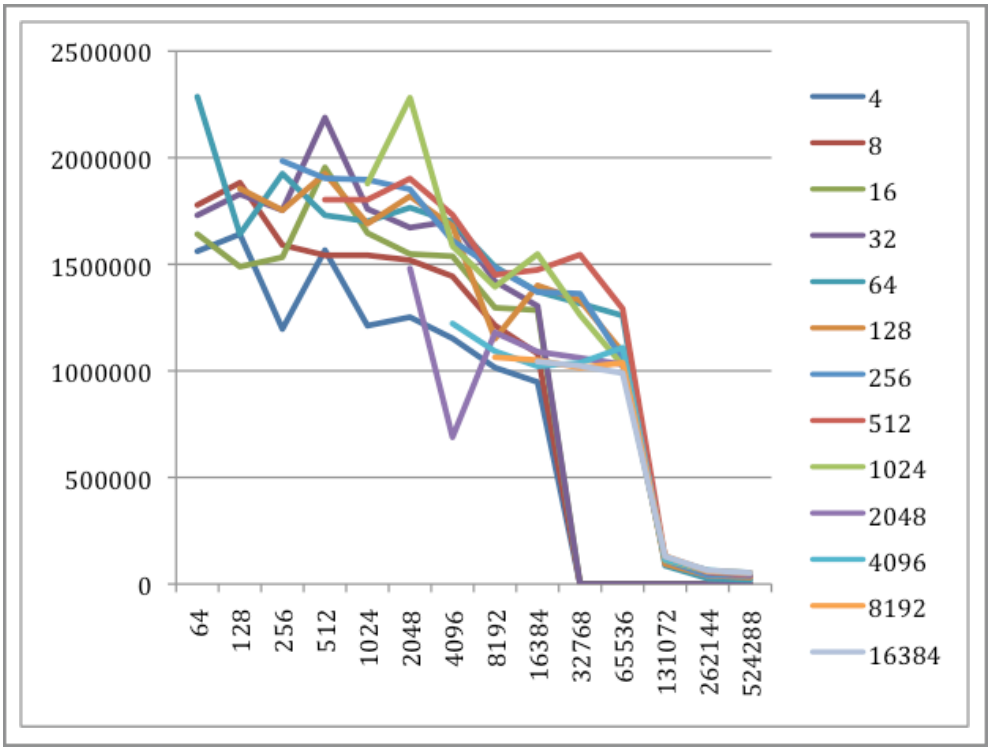


Jaula

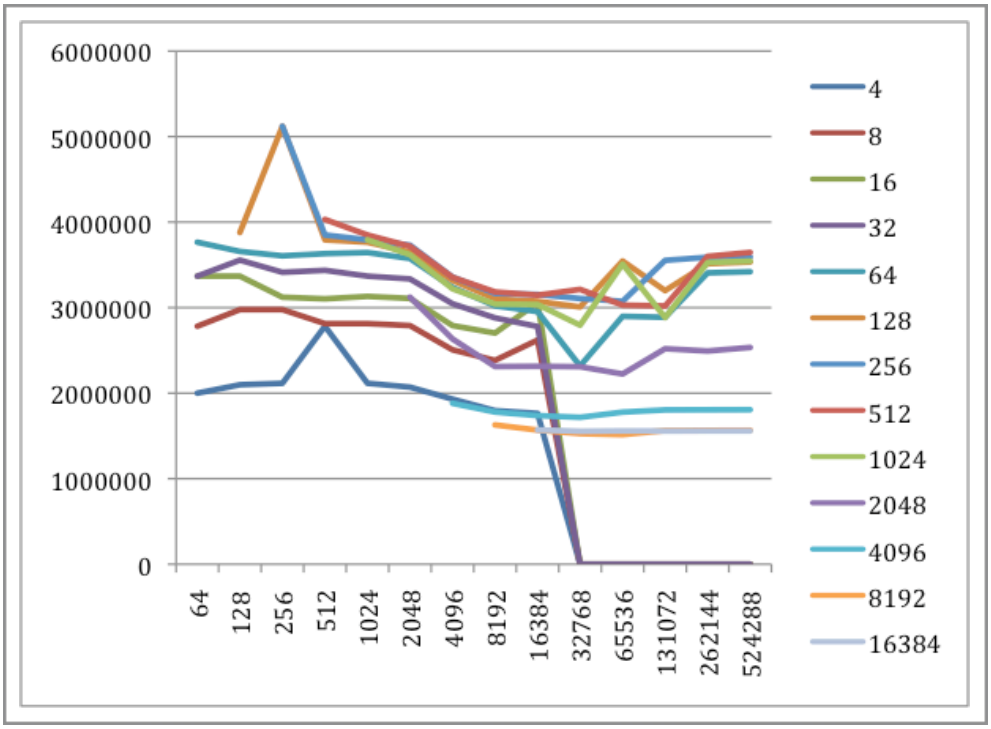
Al igual que en la prueba de escritura, se puede apreciar que en lectura no hay casi pérdida de rendimiento, incluso, esta es menos que en escritura siendo de nuevo apreciable observando las tablas.

Resultados de lectura aleatoria

Sistema operativo



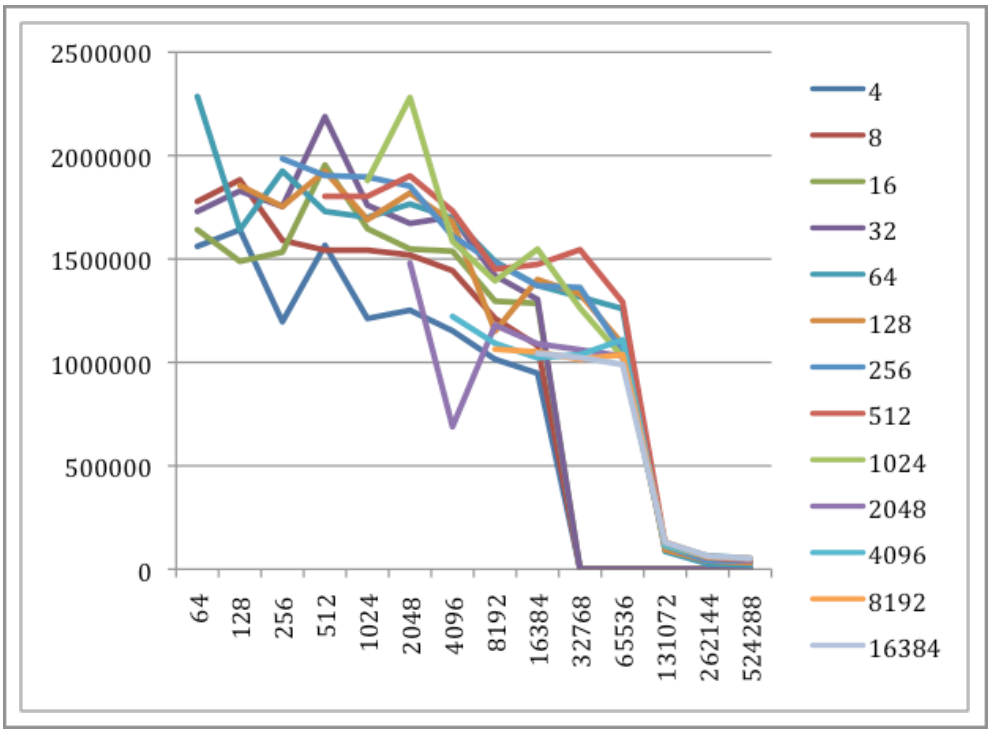
Jaula



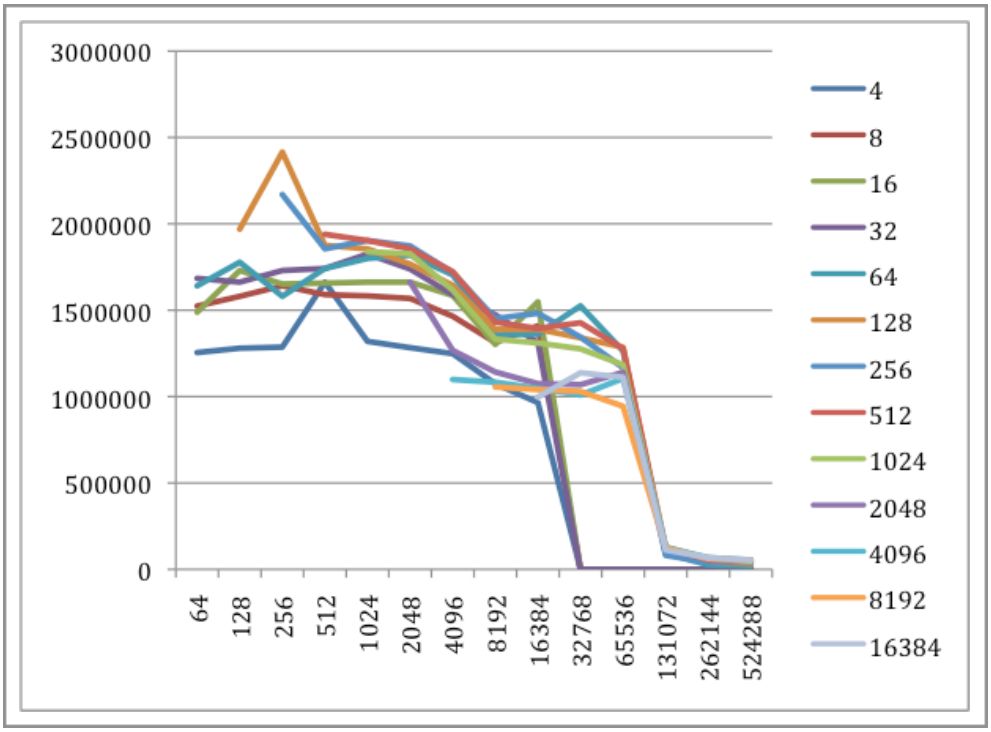
Esta prueba claramente denota una pérdida de rendimiento notable pues se puede observar que el rango de valores de la prueba fuera de la jaula oscila entre 1000000 y 2500000 y dentro de la jaula entre 2000000 y 4000000 pudiendo decir que la pérdida es prácticamente del 100%.

Resultados de escritura aleatoria:

Sistema operativo



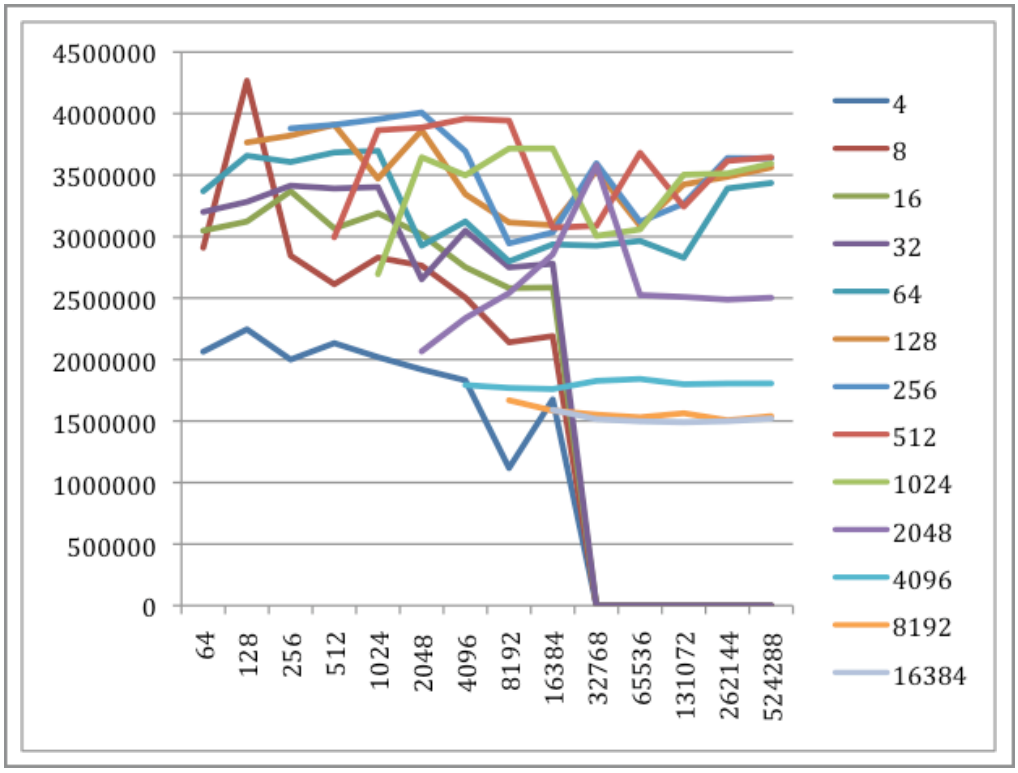
Jaula



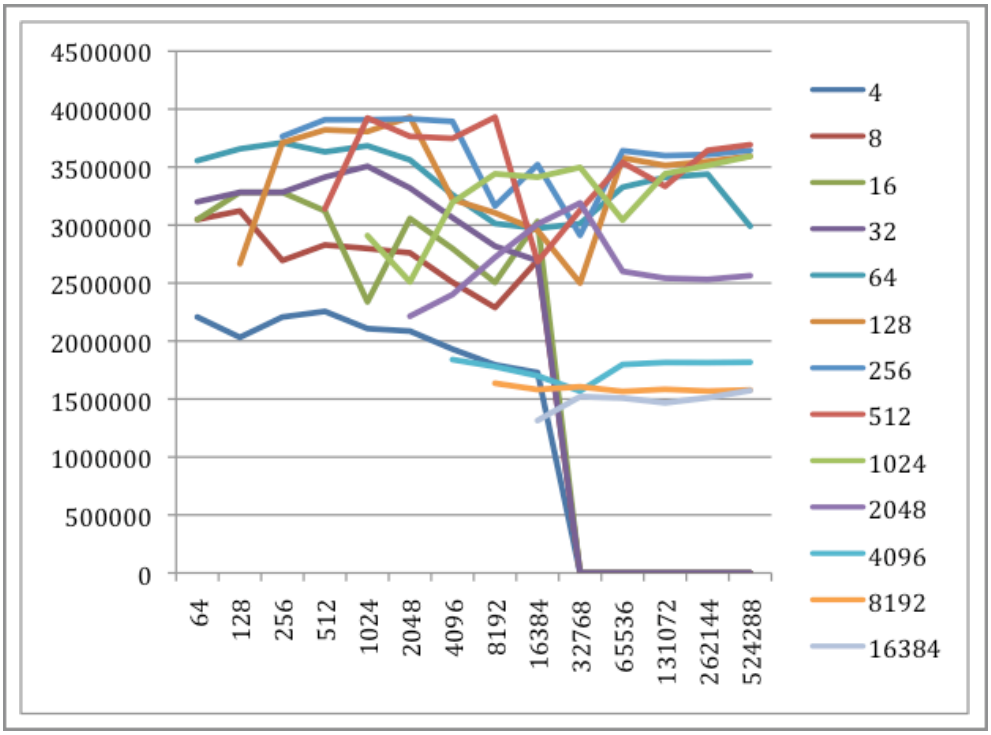
Resulta sorprendente, tras la lectura aleatoria, como en escritura aleatoria no hay casi pérdida de rendimiento sino que llega a tener unos picos de ligera mejoría.

Resultados de lectura con stride(separación entre bloques consecutivos):

Sistema operativo



Jaula



En esta última prueba se puede observar que tampoco ofrece una pérdida de rendimiento acusable sino que esta es muy leve y apreciable a través de las tablas.

Conclusiones del banco de pruebas Iozone

Como conclusión de este banco de pruebas podemos decir, de nuevo, que la pérdida de rendimiento es realmente insignificante que, a pesar de la prueba de lectura aleatoria (que habrá que revisar), se puede considerar como un logro para nuestro proyecto.

Dificultades del proyecto

Además de las propias dificultades inherentes a cualquier desarrollo creemos que dos de ellas han sido las que realmente han marcado el proyecto.

En primer lugar, nuestras mayores dificultades han estado relacionadas con la modificación del núcleo de Linux. Es extenso y complejo, y sólo en fechas cercanas al final hemos comenzado a tener un dominio más amplio de cómo tratar con él.

En segundo lugar: el tiempo. Un proyecto de esta envergadura es necesario planificarlo con un margen mayor de tiempo para abarcar todos los aspectos que hemos detallado a lo largo de nuestra exposición. Sin embargo, sólo hemos alcanzado a realizar un demostrador tecnológico, lo

que es claro símbolo de que la dificultad que percibíamos al principio era menor de la que en realidad era.

Capacidad del sistema

La capacidad de nuestra jaula depende del uso para el que se le haya configurado: servidor, entorno de pruebas, etc. Es capaz de correr casi cualquier sistema bajo consola. Para conseguir ejecutar algún entorno de ventanas hay que realizar una serie de configuraciones que rompen la seguridad del entorno y permiten salir de la jaula ejecutando como un proceso normal, por eso no las hemos incluido.

Se hizo tanto hincapié en el apartado "Preparación del entorno" sobre qué se debe copiar o no a la jaula por este motivo. La máquina virtual es altamente sensible a la configuración previa que se haya efectuado. Depende del usuario, por tanto, dejar el estado del contenedor adaptado a sus necesidades.

Limitación del modelo de virtualización a nivel de SO

Este modelo posee dos limitaciones importante: por un lado la inexistencia de hipervisor y por el otro la compartición del kernel por todas las máquinas virtuales.

Lo primero porque nos deja en una situación de cierta incapacidad para controlar lo que ocurre dentro de la máquina virtual.

Lo segundo, porque no existe un verdadero aislamiento. Las llamadas al sistema de los programas acceden realmente al código del sistema operativo. Si tuviéramos una máquina virtual verdadera, esto no ocurriría o al menos nos permitiría controlarlo (mediante el hipervisor que hemos comentado antes). Sin embargo no ocurre así.

Necesidad de un acceso a red seguro

Uno de los mayores problemas que tienen nuestros contenedores ahora mismo es que para dotarles de acceso a Internet hay que montar explícitamente el directorio */dev* y acabamos con la seguridad del sistema que obtiene por medio de la llamada a **chroot()**. En un entorno de pruebas como es el que presentamos carece de importancia, pero si se quiere elevar el sistema a un nivel profesional es necesario tener capacidad de acceso a red de forma segura.

Algunas de las opciones que nos planteamos ya fueron comentadas en el apartado de "Soluciones descartadas", pero hay muchas más. Siguiendo los referentes que hemos seguido a lo largo de todo el desarrollo, FreeBSD tiene integrada el acceso a la red de forma segura en el propio kernel y FreeVPS usa redes privadas virtuales (VPN). Cualquiera de ambas soluciones es factible.

Conclusión

En primer lugar hemos situado en contexto al lector hablando sobre las distintas tecnologías que hemos usado o valorado, tanto a nivel de hardware(ARM) como de software(VLX, Xen, distintos sistemas operativos). Debemos destacar el gran futuro que tienen todas estas tecnologías: ARM acaba de anunciar que la próxima generación de procesadores Cortex será de doble núcleo y, aunque su orientación comercial se centre en los teléfonos móviles, supone un gran avance en la capacidad de procesamiento de los sistemas empotrados; por su parte, VirtualLogix se está convirtiendo en una compañía de referencia en este sector. Los sistemas operativos en tiempo real han crecido al mismo tiempo que los procesadores que se usan habitualmente en entornos embebidos y son bastante capaces sin renunciar a su principal objetivo: la eficiencia.

En este contexto de búsqueda de eficiencia hemos planteado nuestro proyecto. Hemos hablado sobre el modelo de virtualización que hemos seguido: la virtualización a nivel de sistema operativo. Explicando este modelo hemos hecho un recorrido por todas las opciones que se nos plantearon a lo largo del curso, indicando las que han sido descartadas y las que se han desarrollado. Todos estos modelos han sido explicados con el objetivo de hacer más comprensible nuestra decisión.

Finalmente, hemos detallado la implementación de nuestro modelo. Tanto las modificaciones del sistema operativo como la inicialización de las jaulas han sido expuestas. Se optó por una implementación cuidadosa y respetuosa con los principios de implementación con los que el propio kernel de Linux cuenta.

Nuestro modelo puede ser criticable, pues no es un desarrollo muy extenso, y aun así, los objetivos que nos planteamos los hemos cumplido: **trasparencia** y **eficiencia**. Comparado con otras soluciones hemos obtenido mejor resultado que si hubiéramos implementado un modelo de paravirtualización como era la idea inicial. Sin embargo, falta mucho trabajo por delante para alcanzar el nivel de implementaciones comerciales de Linux VServer o Free BSD Jails, que tienen varios años de desarrollo a sus espaldas, como fue el objetivo posterior.

Sin embargo, comparando con las primeras versiones de Linux VServer, hemos visto que hemos tenido comienzos similares en lo referente a las ideas que hemos planteado. Por supuesto, a través de la experiencia y con la dedicación necesaria se pueden obtener innumerables mejoras que elevarían la calidad de nuestro proyecto al nivel de estas implementaciones.

Al comenzar el proyecto nos planteamos diversos retos que hemos visto superados en mayor o menor medida: la modificación del kernel de Linux, la comprensión del mecanismo de llamadas al

sistema, el uso adecuado del chroot y la preparación del entorno de la jaula, el análisis de las pruebas de eficiencia, etc. Todos estos aspectos nos han proporcionado, en nuestra opinión, una gran experiencia que valoramos de gran utilidad para el futuro cuando desarrollemos e investiguemos en proyectos comerciales.

En definitiva, la mayor complejidad de este proyecto ha sido el aprendizaje necesario para enfrentarnos, entre los ya citados aspectos, a la modificación del núcleo de Linux. Esta ha resultado ser la tarea más compleja y sin duda, la mayor parte del tiempo que hemos dedicado ha sido en esta labor. Además, consideramos que nuestro proyecto es completamente propio pues, bajo la tutela de la profesora, hemos sido capaces de realizar, en mayor o en menor medida, una idea propia, obtenida de una investigación previa al comienzo del presente curso, que ha evolucionado progresivamente a medida que íbamos comprendiendo la investigación realizada.

Glosario

A) **Virtualización: se define como la capacidad de ejecutar una aplicación de un sistema sobre otro con trasparencia para el usuario.**

B) **Paravirtualización: es un modelo de virtualización en el cual se modifica el sistema operativo "guest" para hacer uso de una API de llamadas al sistema adaptada.**

C) **Xen: producto desarrollado por Citrix Systems que usando paravirtualización, modifica el sistema operativo (habitualmente Linux) para que sea capaz de actuar como "host" y permitir ejecutar otros sistemas operativos como "guest"(clientes).**

D) **ARM: se conoce así al conjunto de arquitecturas desarrolladas por la empresa de nombre homónimo.**

E) **Vmware: empresa desarrolladora de un conjunto de productos de virtualización para sobremesa y servidor. Sus productos están disponibles para varios sistemas operativos.**

F) **Sistema operativo en tiempo real: sistema operativo desarrollado para aplicaciones en tiempo real, con un bajo tiempo de respuesta. Suelen caracterizarse por usar poca memoria, ocupar poco espacio y dar soporte para un conjunto de eventos pequeño.**

G) **Virtualización a nivel de sistema operativo: modelo de virtualización que evita la sobrecarga de otros modelos de virtualización modificando el sistema operativo (habitualmente Linux) para permitir ejecutar instancias o subinstancias de si mismo donde se agrupen las aplicaciones (denominadas habitualmente "jaulas" o "contenedores").**

H) **Chroot: llamada al sistema que se encuentra en la mayoría de los sistemas operativos de la familia UNIX. Permite cambiar el directorio raíz del que parte la aplicación que realiza la llamada.**

I) **Contenedor o jaula: modelo de desarrollo de una máquina virtual que se realiza a nivel de sistema operativo basado en la funcionalidad que ofrece la llamada Chroot.**

J) **Free BSD Jails: implementación de virtualización a nivel de sistema operativo para FreeBSD que se encuentra integrada en el propio kernel de FreeBSD.**

K) **Encriptación: proceso de volver ilegible información mediante diversos mecanismos, así como después volver a convertir la información en legible.**

L) **Nbench: banco de pruebas basado en diversos ámbitos como los algoritmos utilizados más comúnmente en el sistema y operaciones en punto flotante.**

M) **Iozone: banco de pruebas basado en operaciones de entrada/salida sobre lectura y escritura de información y obteniendo como resultado el número de operaciones.**

N) **IPC (*Inter Process Communication*): recursos que permiten la comunicación y sincronización entre procesos. Está formado por semáforos, memoria compartida y mensajes.**

Bibliografía

James E. Smith, Ravi Nair. “Virtual machines: versatile platforms for systems and processe”. San Francisco: Morgan Kaufman, cop. 2005

Andrew Sloss, Dominic Symes, Chris Wright; whit a contribution by John Rayfield. “ARM system developer's guide: designing and optimizing system software”. San Francisco, CA: Elsevier/ Morgan Kaufman, cop. 2004

Robert Love. “Linux Kernel Development, Second Edition”. Novell Press 2005

Robert Love. “Linux System Programming, 1st Edition”. O'Reilly Media, Inc. 2007

Daniel P. Bovet; Marco Cesati. “Understanding the Linux Kernel, 3rd Edition”. O'Reilly Media, Inc. 2007

Michael Barr; Anthony Massa. “Programming Embedded Systems, 2nd Edition”. O'Reilly Media, Inc. 2006

Philippe Gerum; Karim Yaghmour; Jon Masters; Gilad Ben-Yossef. “Building Embedded Linux Systems, 2nd Edition”. O'Reilly Media, Inc. 2008

Anthony J. Massa. “Embedded Software Development With ECOS”. Prentice Hal 2002

Jeanna N. Matthews; Eli M. Dow; Todd Deshane; Wenjin Hu; Jeremy Bongio; Patrick F. Wilbur; Brendan Johnson. “Running Xen: A Hands-On Guide to the Art of Virtualization”. Prentice Hall 2008

Links de referencia

- http://www.uclinux.org/
- http://www.debian.org/ports/arm/
- http://www.trango-vp.com/documentation/technical.php
- http://www.virtuallogix.com/products/what-is-virtualization.html
- http://www.ok-labs.com/products/okl4
- http://www.movingtofreedom.org/2007/02/21/howto-encfs-encrypted-file-system-in-ubuntu-and-fedora-gnu-linux/
- http://beagleboard.org/hardware